Using HPCToolkit to Measure and Analyze the Performance of CPU and GPU-accelerated Parallel Applications

John Mellor-Crummey
Rice University

ATPESC 2021
August 11, 2021

Download application examples to run, measure, and analyze on ascent
git clone https://github.com/HPCToolkit/hpctoolkit-tutorial-examples

See theta:/grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit
(contains an example to run and analysis as well as measured data to explore)

Project Overview

- **Goals**
  - Develop effective tools to measure, attribute, analyze, and understand performance of applications, libraries, tools, and system software on extreme-scale parallel systems
  - Develop new strategies and mechanisms to measure and analyze performance and resource utilization of GPU-accelerated compute nodes
  - Lead evolution of hardware and software ecosystems to enable better tools
- **Team**
  - Lead Institution: Rice University (HPCToolkit performance tools)
    - PI: Prof. John Mellor-Crummey
    - Research staff: Dr. Laksono Adhianto, Dr. Mark Krentel, Dr. Xiaozhu Meng, Dr. Scott Warren
    - Grad students: Keren Zhou, Jonathon Anderson, Yumeng Liu, Aaron Cherian, Dejan Grubisic
  - Subcontractor: University of Wisconsin – Madison (Dyninst binary analysis toolkit)
    - Lead: Prof. Barton Miller
- **Principal Funding**:
  - DOE Exascale Computing Project, ANL, NNSA Tri-labs, AMD, Intel

Performance Analysis Challenges on Modern Supercomputers

- **Myriad performance concerns**
  - Computation performance on CPU and GPU
  - Data movement costs within and between memory spaces
  - Internode communication
  - I/O
- **Many ways to hurt performance**
  - insufficient parallelism, load imbalance, serialization, replicated work, parallel overhead …
- **Hardware and execution model complexity**
  - Multiple compute engines with vastly different characteristics, capabilities, and concerns
  - Multiple memory spaces with different performance characteristics
    - CPU and GPU have different complex memory hierarchies
  - Often, a large gap between programming model and implementation
    - e.g., OpenMP, template-based programming models
  - Asynchronous execution

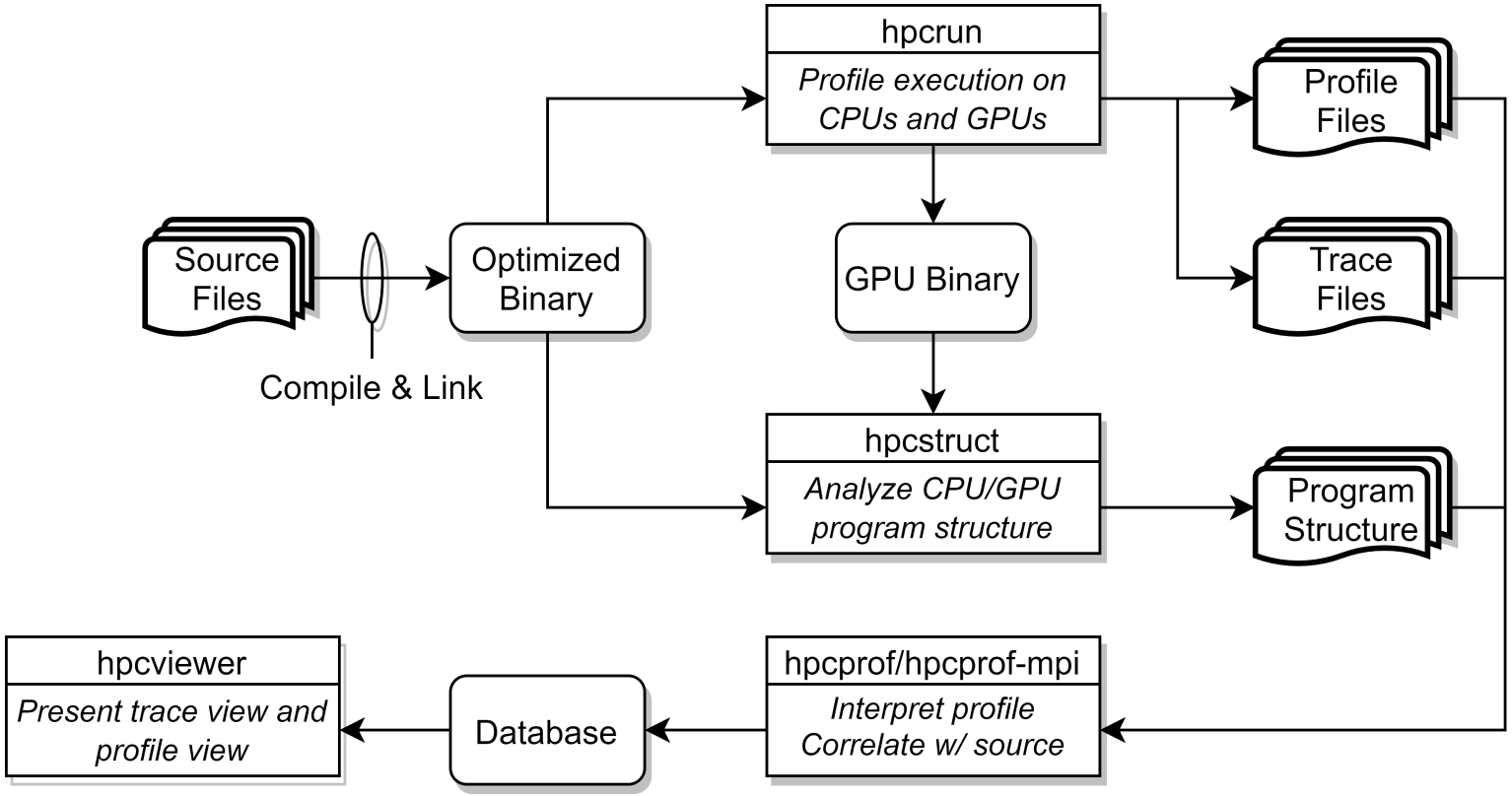
Outline

- **Overview of Rice's HPCToolkit**
- **Understanding the performance of parallel programs using HPCToolkit's GUIs**
  - code centric views
  - time centric views
- **Monitoring GPU-accelerated applications**
- **Work in progress**

Rice University's HPCToolkit Performance Tools

- **Employs binary-level measurement and analysis**
  - Observes executions of fully optimized, dynamically-linked applications
  - Supports multi-lingual codes with external binary-only libraries
- **Collects sampling-based measurements of CPU**
  - Controllable overhead
  - Minimize systematic error and avoid blind spots
  - Enable data collection for large-scale parallelism
- **Measures GPU performance using APIs provided by vendors**
  - Callbacks to monitor launch of GPU operations
  - Activity API to monitor and present information about asynchronous operations on GPU devices
  - PC sampling for fine-grain measurement
- **Associates metrics with both static and dynamic context**
  - Loop nests, procedures, inlined code, calling context on both CPU and GPU
- **Specify and compute derived CPU and GPU performance metrics of your choosing**
  - Diagnosis often requires more than one species of metric
- **Supports top-down performance analysis**
  - Identify costs of interest and drill down to causes: up and down call chains, over time

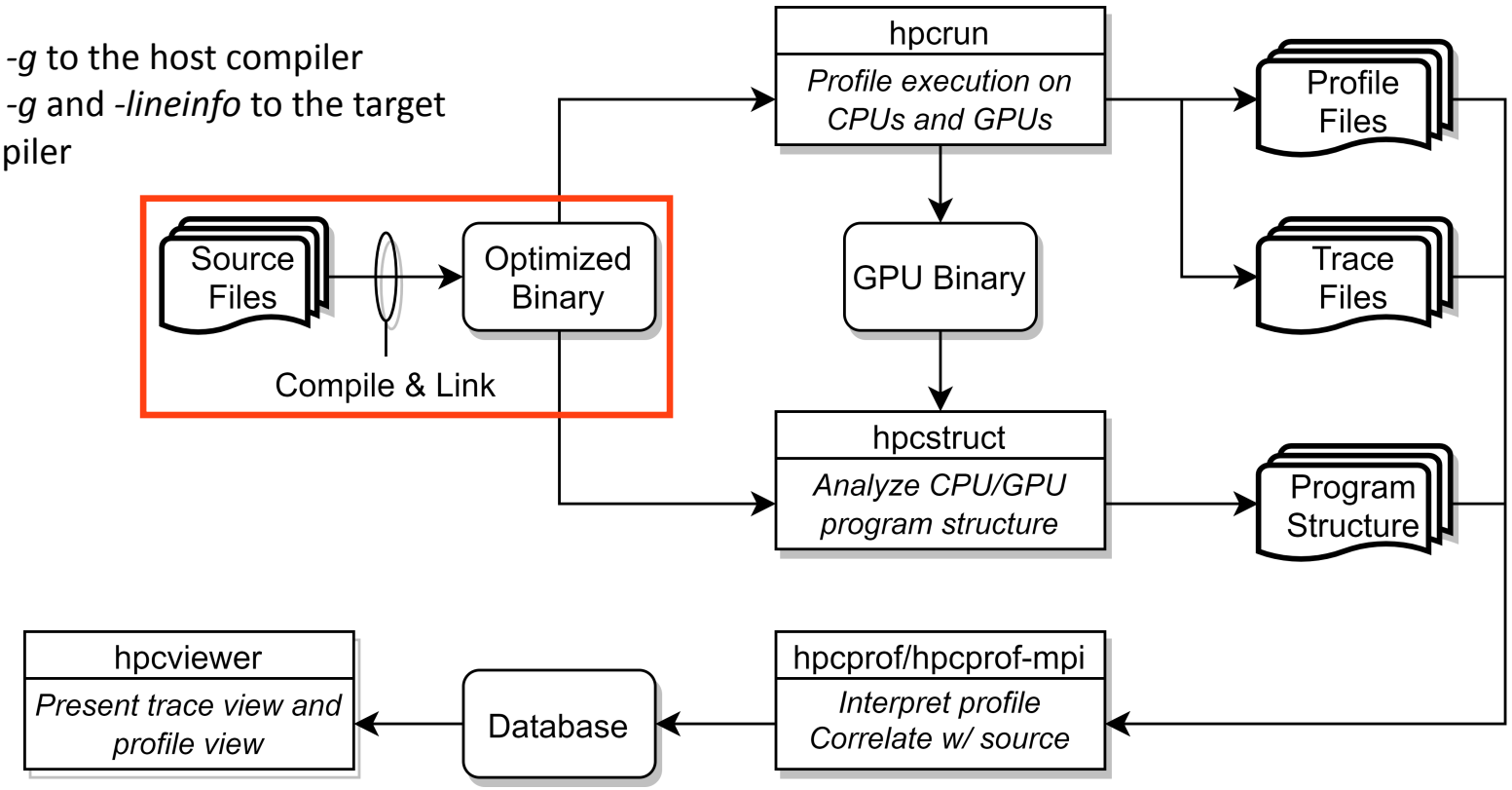
HPCToolkit's Workflow for GPU-accelerated Applications



HPCToolkit's Workflow for GPU-accelerated Applications

Step 1:

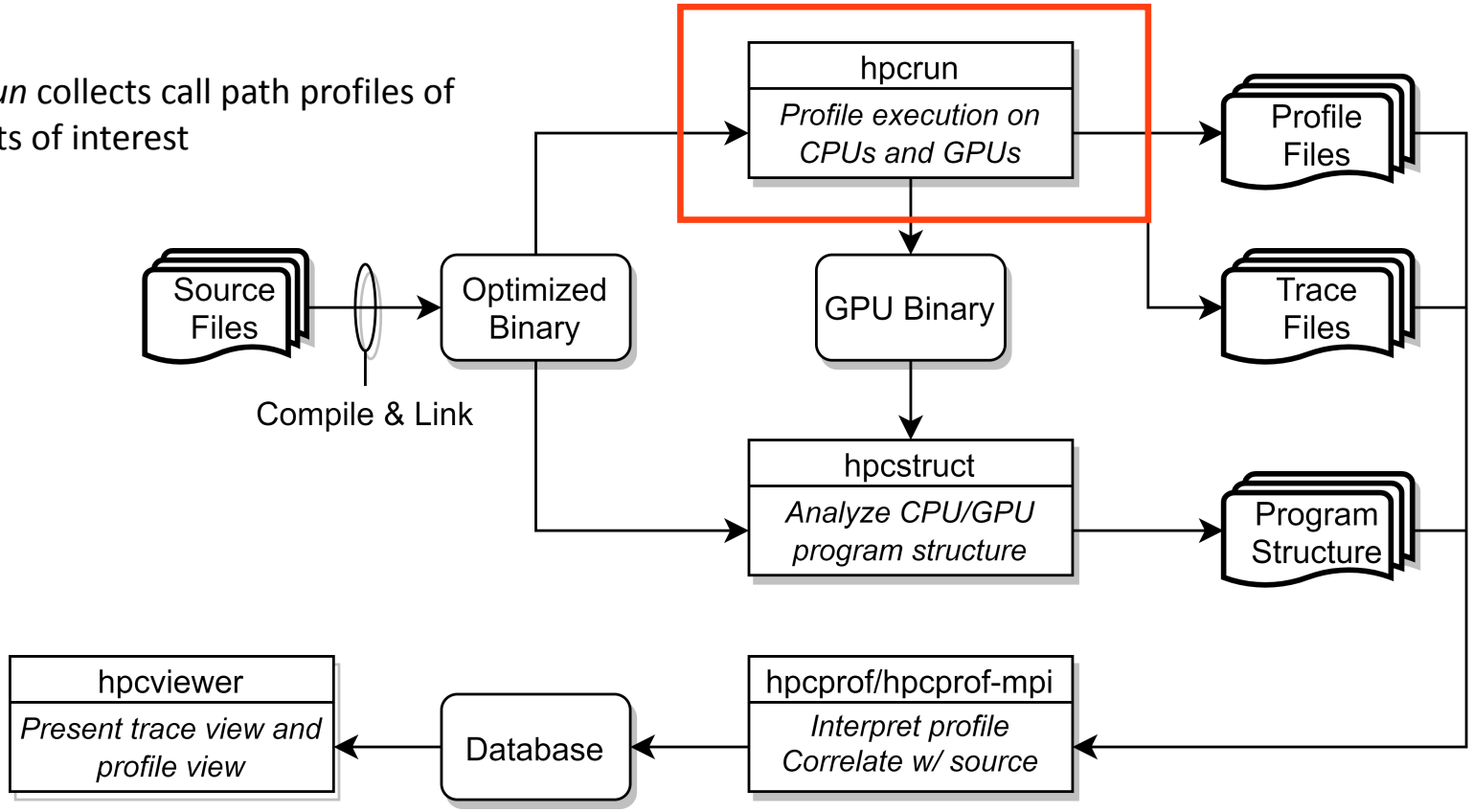
- Add *-g* to the host compiler
- Add *-g* and *-lineinfo* to the target compiler



HPCToolkit's Workflow for GPU-accelerated Applications

Step 2:

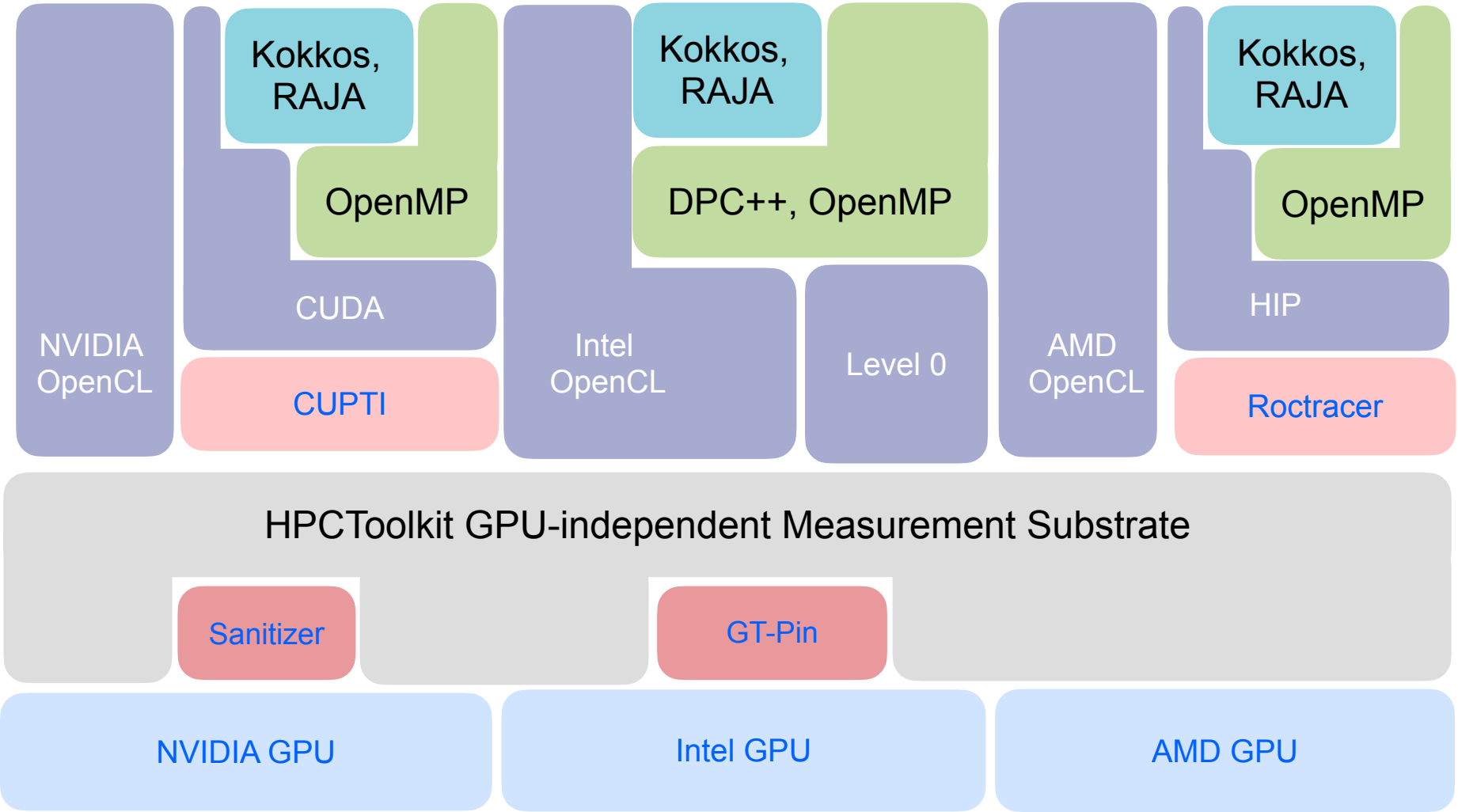
- *hpcrun* collects call path profiles of events of interest



hpcrun - Measure CPU and GPU execution

- GPU profiling
  - hpcrun -e gpu=xxx <app> …. // *xxx* ∈ *{nvidia,amd,opencl,level0}*
- GPU tracing (-t)
  - hpcrun -e gpu=yyy -t <app> // *yyy* ∈ *{nvidia,amd,opencl}*
- GPU PC sampling (NVIDIA only)
  - hpcrun -e gpu=nvidia,pc -t <app>
- CPU and GPU profiling and tracing
  - hpcrun -e REALTIME -e gpu=yyy -t <app>
- Use hpcrun with job launchers
  - jsrun -n 32 -g 1 -a 1 hpcrun -e gpu=*xxx* <app>
  - srun -n 1 -G 1 hpcrun -e gpu=*xxx* <app>
  - aprun -n 16 -N 8 -d 8 hpcrun -e gpu=*xxx* <app>

HPCToolkit and GPU Software Stacks



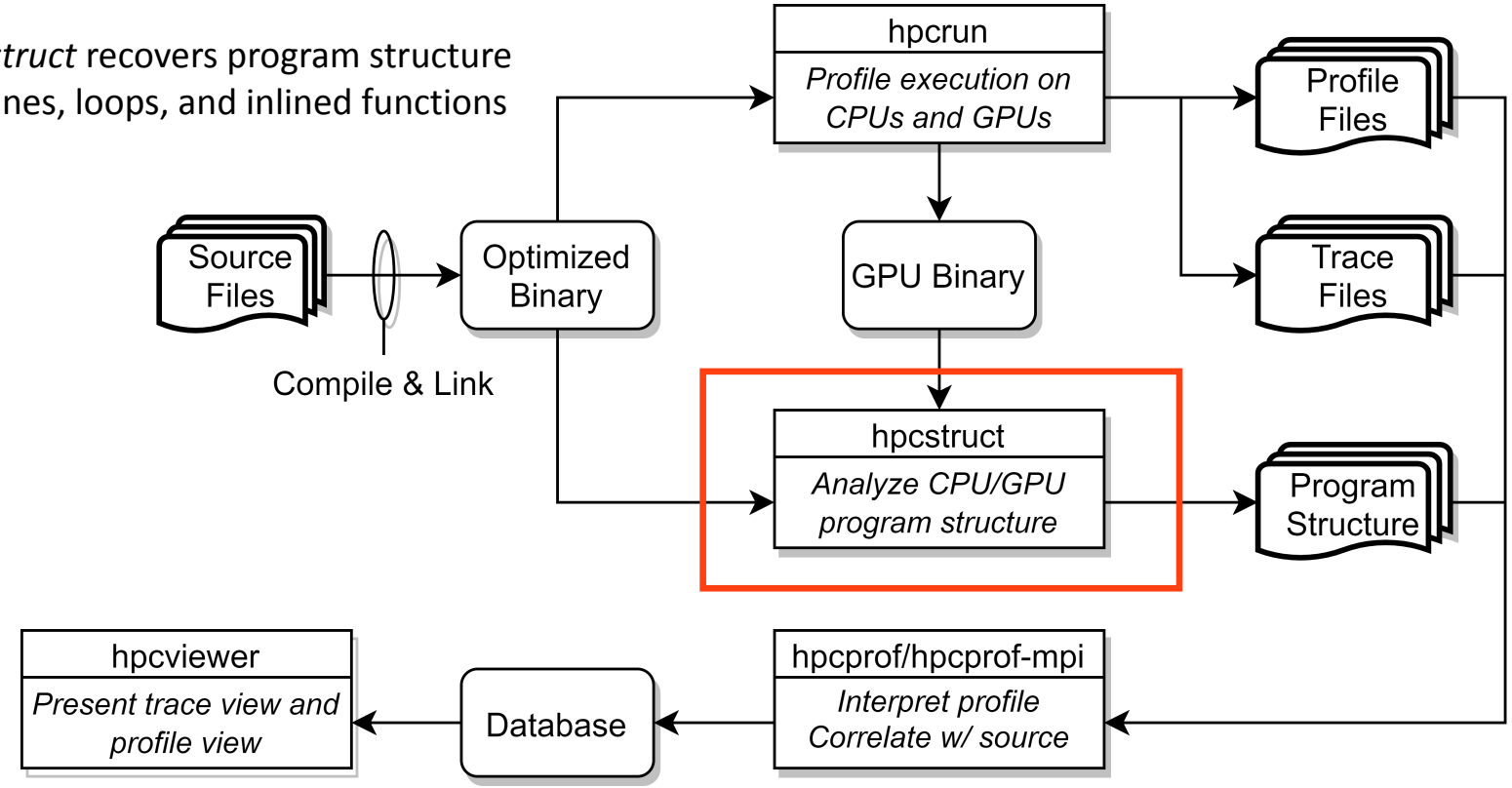
Measurement for GPU-accelerated Supercomputers

- **Measurement interfaces**
  - Hardware
    - CPU hardware performance monitoring unit
    - GPU hardware counters and PC sampling
  - Software
    - Glibc LD_AUDIT for tracking dynamic loading of shared libraries
    - Linux perf_events for kernel measurement
    - GPU monitoring and instrumentation libraries from vendors
- **Multiple measurement modalities and interfaces**
  - Sampling on the CPU
  - Callbacks when GPU operations are launched and (sometimes) completed
  - GPU event stream, including PC sampling measurements

HPCToolkit's Workflow for GPU-accelerated Applications

Step 3:

- *hpcstruct* recovers program structure about lines, loops, and inlined functions



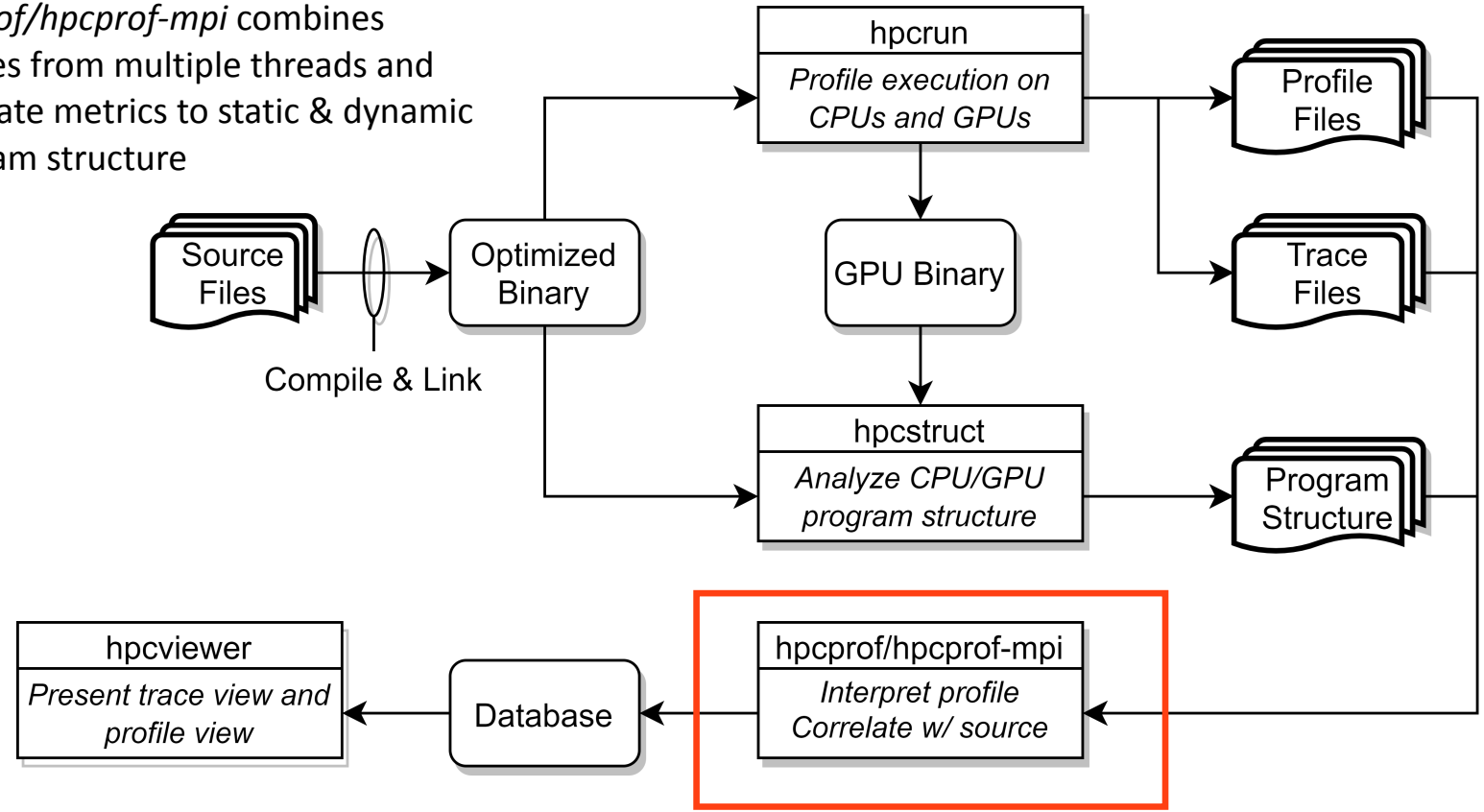
hpcstruct - Recover Program Structure

- Analyze CPU binaries
  - hpcstruct [-j <threads>] <app>
  - yields a file <app>.hpcstruct
- Analyze all GPU binaries in <measurements-dir>
  - hpcstruct [-j <threads>] [--gpucfg yes] <measurements-dir>
    - “gpucfg yes” means recover GPU loop nests, calling context information
  - augments the measurement directory with a hpcstruct file for each GPU binary

HPCToolkit's Workflow for GPU-accelerated Applications

Step 4:

- *hpcprof/hpcprof-mpi* combines profiles from multiple threads and correlate metrics to static & dynamic program structure



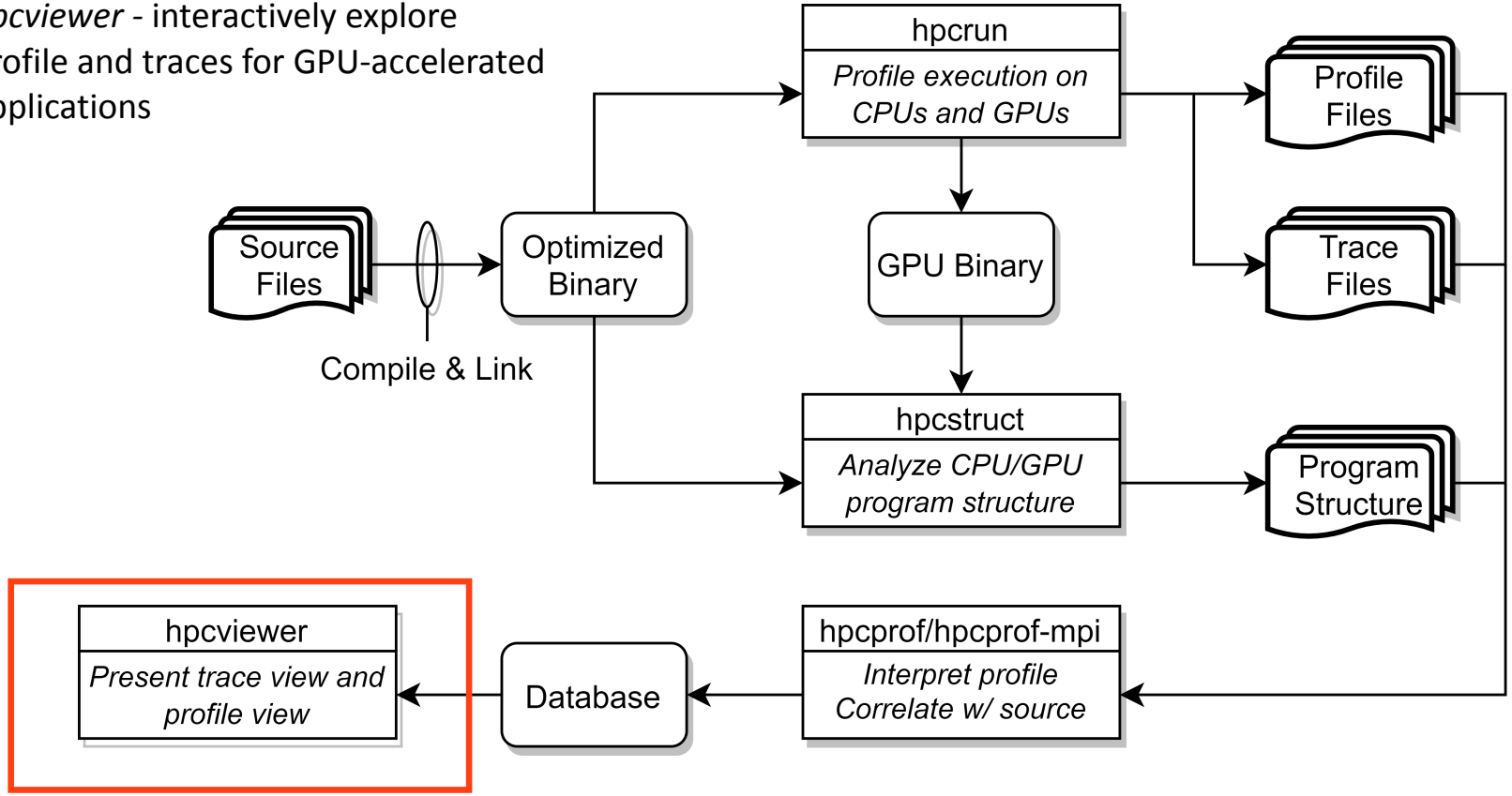
hpcprof/hpcprof-mpi - Correlate Measurements with Code

- Use a single process to combine performance data
  - hpcprof -S <app>.hpcstruct <measurements-dir>
- Use multiple processes to combine performance data
  - jsrun -n <np> hpcprof-mpi -S <app>.hpcstruct <measurements-dir>
  - srun -n <np> hpcprof-mpi -S <app>.hpcstruct <measurements-dir>

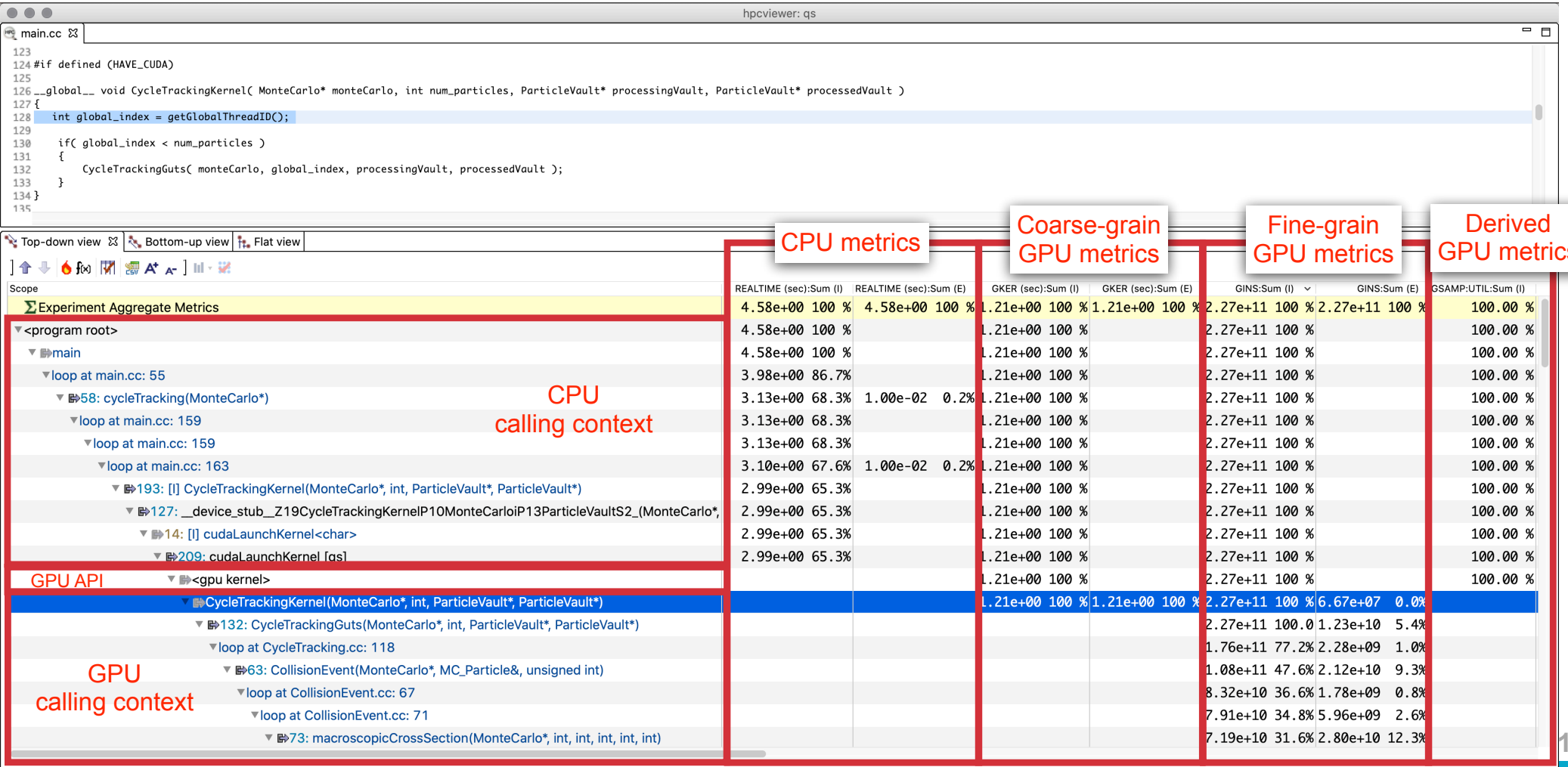
HPCToolkit's Workflow for GPU-accelerated Applications

Step 4:

- *hpcviewer* - interactively explore profile and traces for GPU-accelerated applications



HPCToolkit's Code-Centric Profiles of GPU-accelerated Code

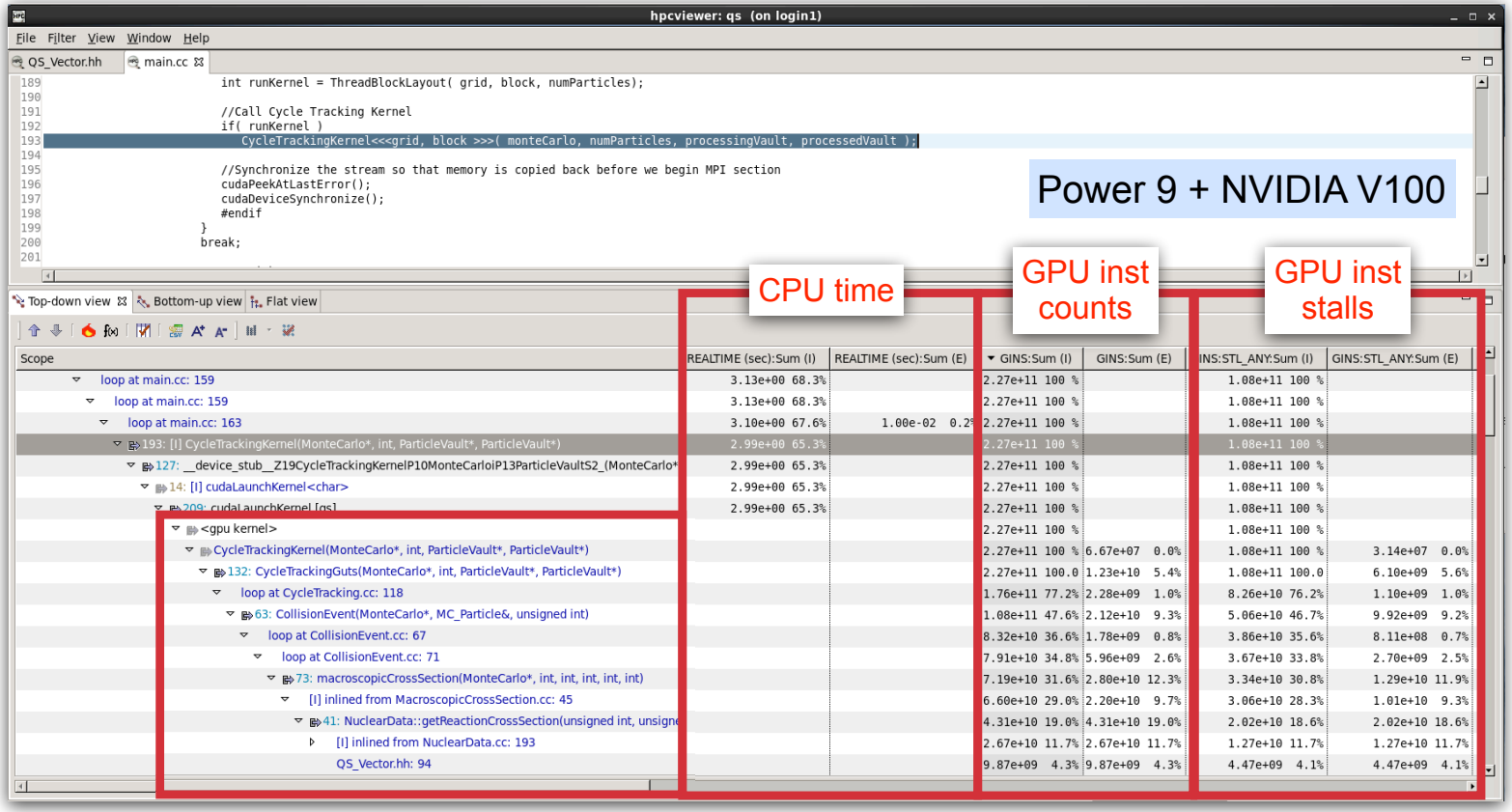


Coarse- and Fine-grain Measurement on NVIDIA GPUs: ECP Quicksilver

Compute Node

- 2xPower9 + 4xNIVIDIA GPUs

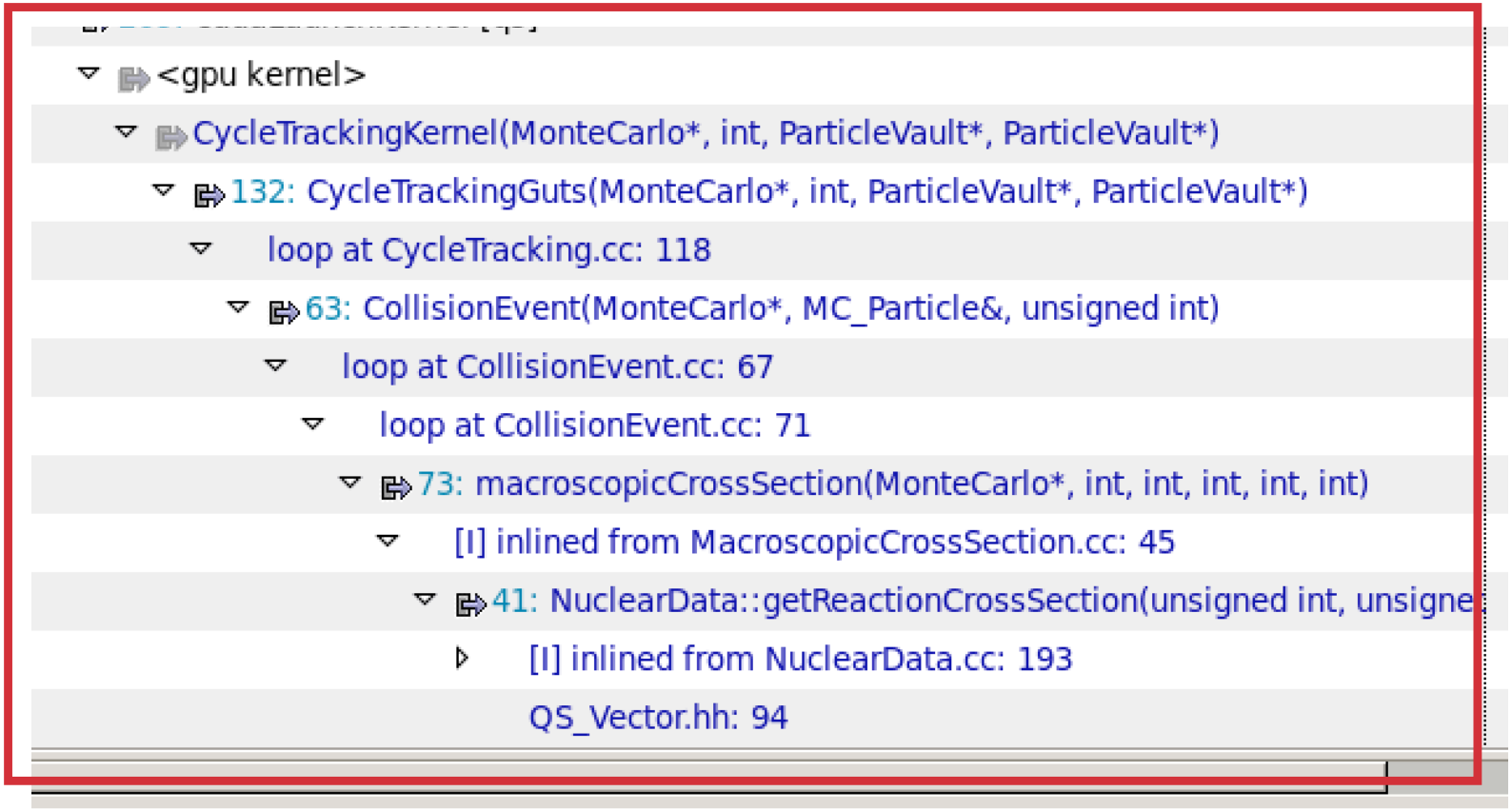
- **Optimized (-O2) compilation with nvcc**
- **Detailed measurement and attribution using PC sampling**
- **Reconstruct approximate GPU calling context tree from flat PC samples**
- **Understand GPU loops and inlined code**
- **Attribute information to heterogeneous calling context**
- **Key Metrics**
  - instructions executed
  - instruction stalls and reasons
  - GPU utilization



K. Zhou, M. W. Krentel, and J. Mellor-Crummey. Tools for top-down performance analysis of GPU-accelerated applications. International Conference on Supercomputing. ACM, New York, NY, USA, June, 2020.

Coarse- and Fine-grain Measurement on NVIDIA GPUs: ECP Quicksilver

- **HPCToolkit reconstructs approximate GPU calling context tree from flat PC samples**
- **Understand GPU loops and inlined code**
- **Attribute information to heterogeneous calling context**



K. Zhou, M. W. Krentel, and J. Mellor-Crummey. Tools for top-down performance analysis of GPU-accelerated applications. International Conference on Supercomputing. ACM, New York, NY, USA, June, 2020.

Coarse- and Fine-grain Measurement on Intel GPUs: ECP PeleC

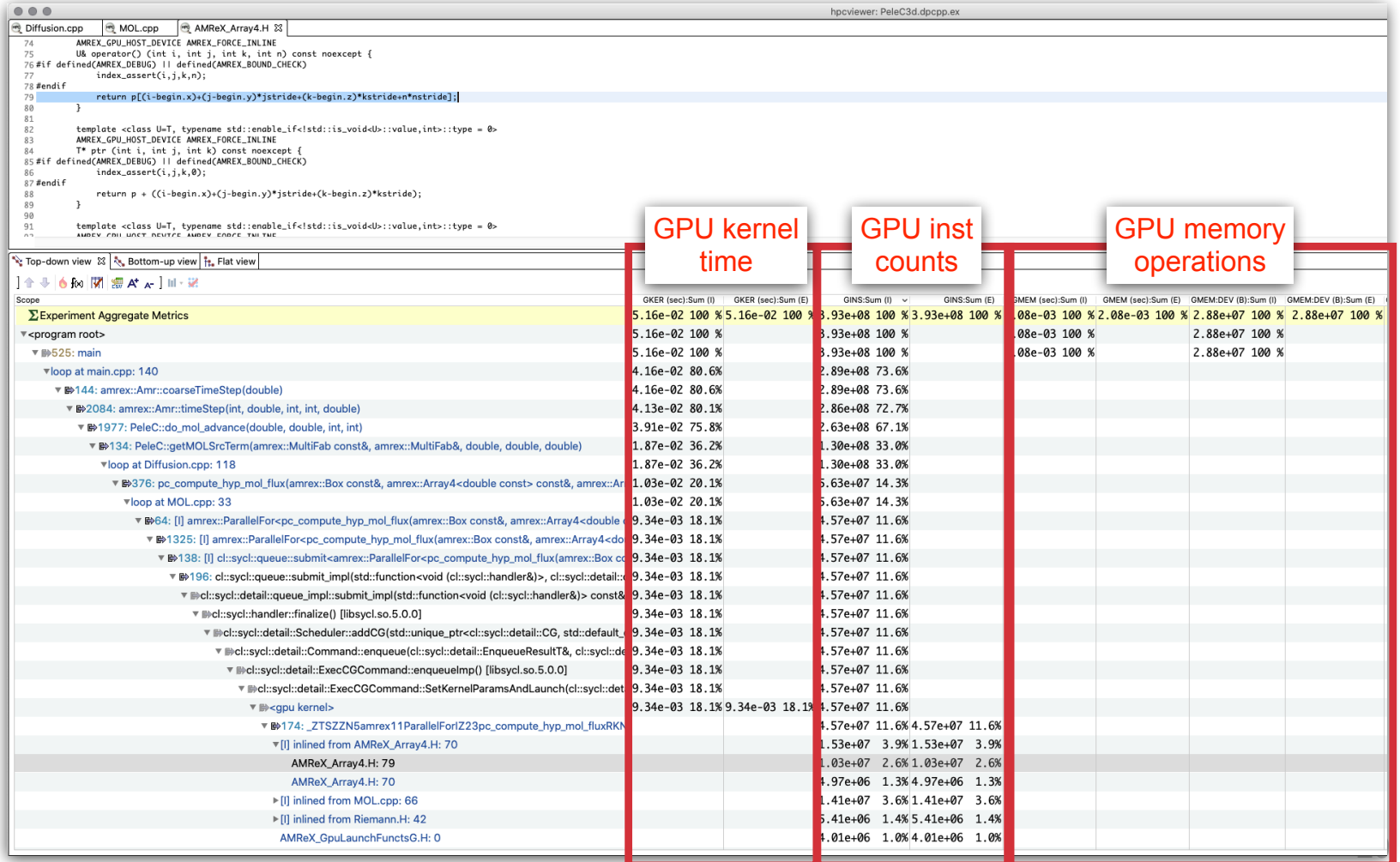
Hardware: JLSE Iris

- Intel E3-1585 v5
- Intel Iris Pro Graphics P580

Software

- DPC++
- oneAPI beta 10
- OpenCL runtime
- GT-Pin

Permission for open publication granted by Kent Moffatt, Intel



Coarse-grain Measurement on AMD GPUs: ECP PeleC

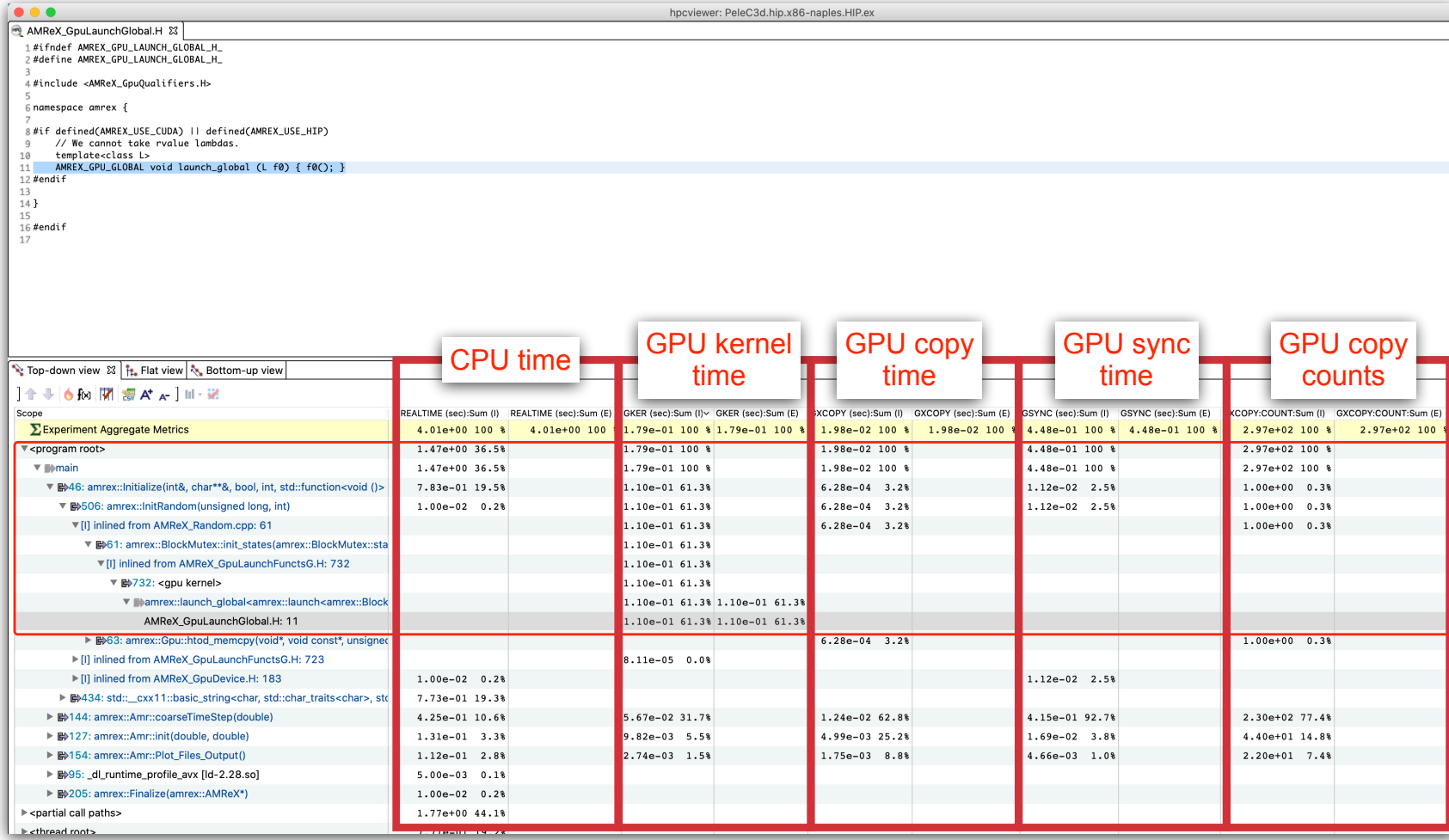
Hardware: Cray Tulip

- AMD Epyc 7601
- 4 x AMD MI60

Software

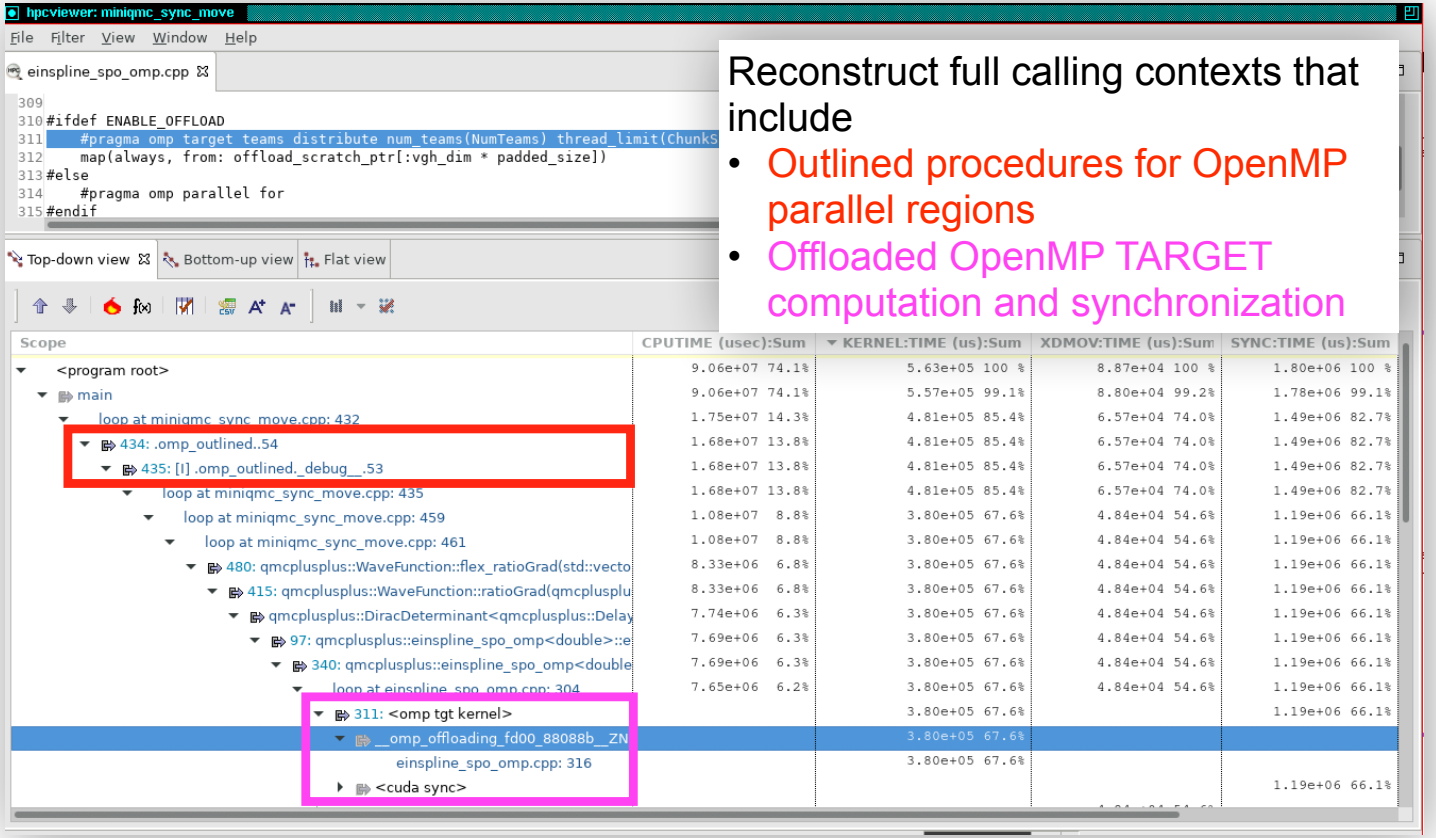
- ROCM 3.8

Permission for open publication granted by Noah Reddell, HPE



Support for OpenMP TARGET: ECP miniqmc

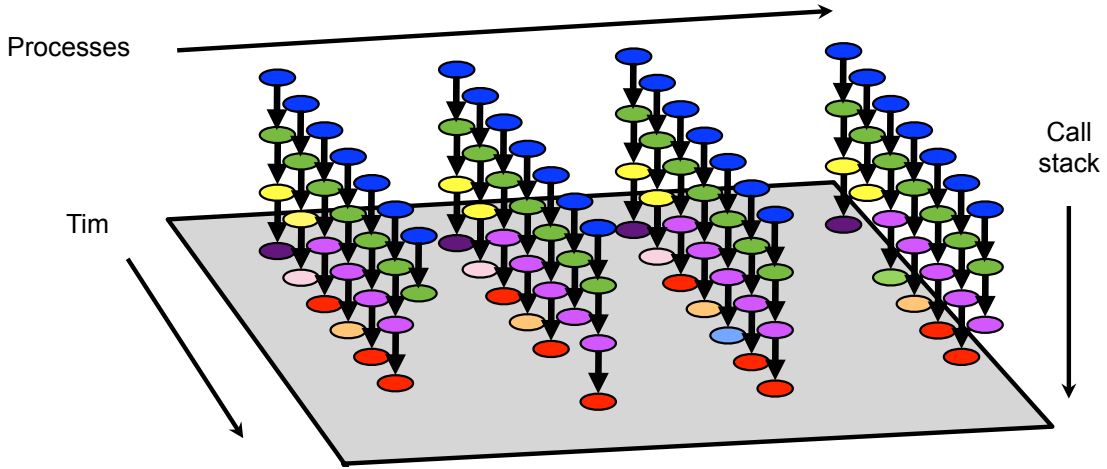
- HPCToolkit implementation of OMPT OpenMP API
  - host monitoring
    - employs OMPT API for call stack introspection
  - GPU monitoring
    - leverages callbacks for device initialization, kernel launch, data operations
  - reconstruction of user-level calling contexts
- Leverages implementation of OMPT in LLVM OpenMP and libomptarget



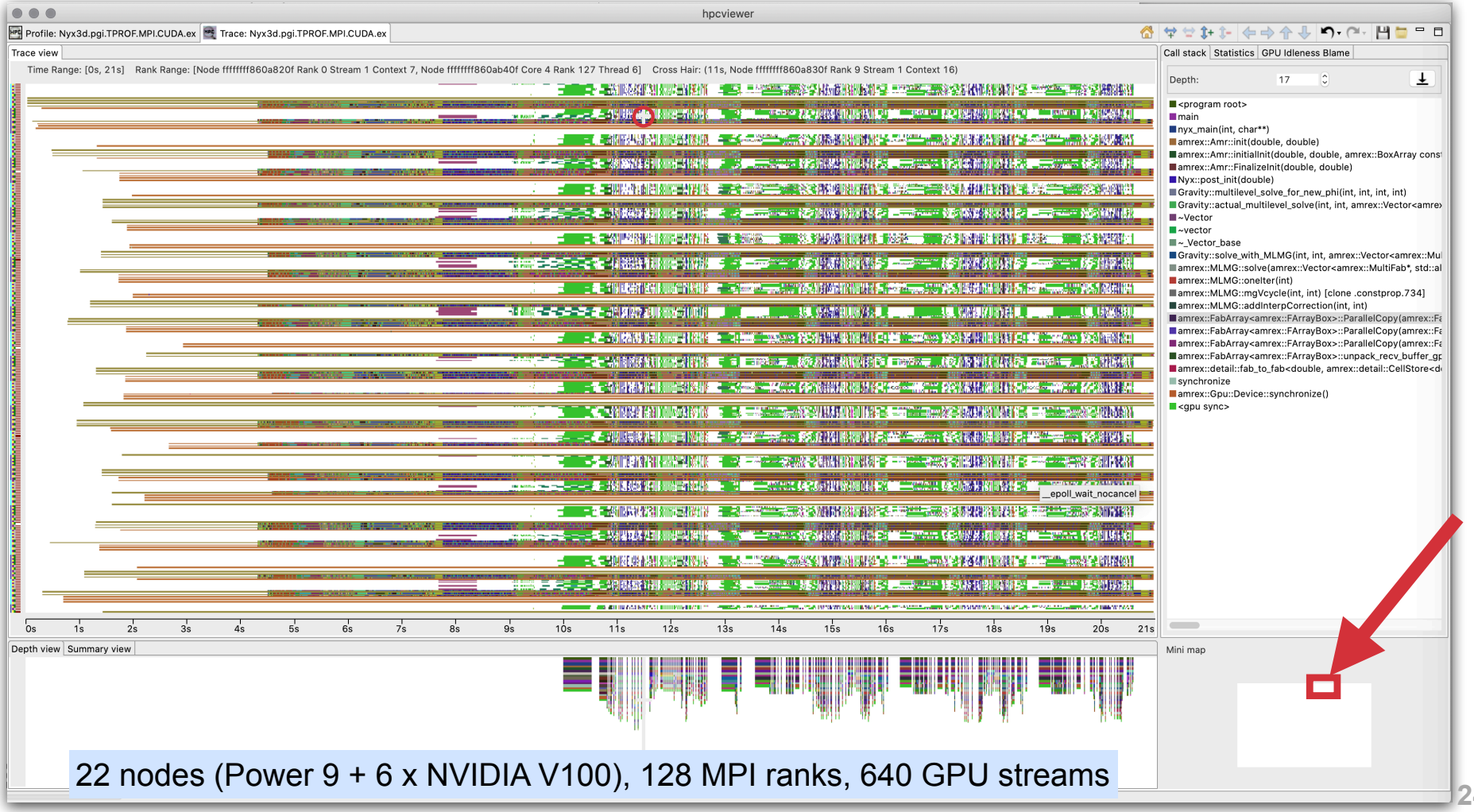
System: Power9 + NVIDIA V100

Understanding Temporal Behavior

- **Profiling compresses out the temporal dimension**
  - Temporal patterns, e.g. serial sections and dynamic load imbalance are invisible in profiles
- **What can we do? Trace call path samples**
  - N times per second, take a call path sample of each thread
  - Organize the samples for each thread along a time line
  - View how the execution evolves left to right
  - What do we view? assign each procedure a color; view a depth slice of an execution

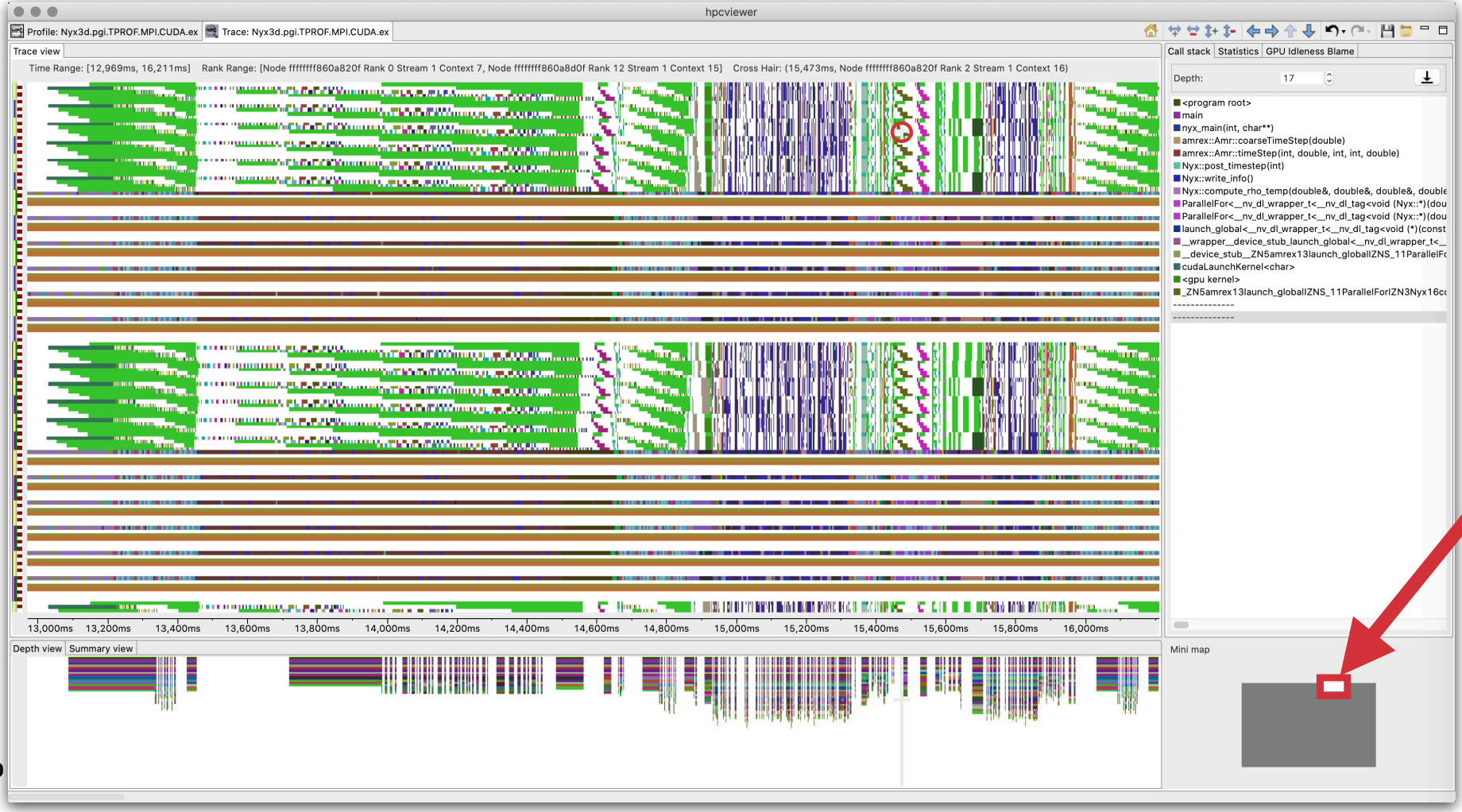


Trace of Multi-rank Multi-GPU Executions: ECP Nyx on Summit



22 nodes (Power 9 + 6 x NVIDIA V100), 128 MPI ranks, 640 GPU streams

Trace of Multi-rank Multi-GPU Executions: ECP Nyx on Summit



GPU Monitoring Capabilities of HPCToolkit

Measurement Capability	NVIDIA	AMD	Intel
kernel launches, explicit memory copies, synchronization	callbacks + activity API	callbacks + Activity API	callbacks
instruction-level measurement and analysis	PC sampling of GPU code	Future*: PC sampling (as seen on Github) of GPU code	GTPin; Future*: instruction-level measurement of GPU code
kernel characteristics	Activity API	(available statically)	(unknown)

Significant support in master branch

Prototype support in master branch

Prototype support in master branch

*Approved for public release

Work in Progress in HPCToolkit

- **GPU Enhancements**
  - Intel GPUs
    - Measurement support for Intel GPUs using OpenCL and Level 0
    - Fine-grain measurement using GTPin
    - Fine-grain attribution using binary analysis
  - AMD GPUs
    - Binary analysis and instrumentation for fine-grain measurement and attribution
  - NVIDIA GPUs
    - Interpret inlining information available (CUDA 11.2)
    - Reduce fine-grain measurement overhead with low-overhead PC sampling (CUDA 11.3)
- **Scalability**
  - Accelerate analysis of measurement data with hpcprof-mpi using multithreading
  - Use sparse formats to reduce size of measurement data and analysis results
- **User interface**
  - Overhauling metric view to enhance performance and scalability
  - Associate trace lines with metadata (node, GPU, MPI rank, GPU stream …)
  - Improve presentation of the many GPU metrics
- **Reliability and Completeness**

Detailed Performance Analysis Requires Support at Many Levels

Hardware and Software Stack Components	Partners
• **Hardware must include support for fine-grain measurement and attribution** • performance counters are not enough; NVIDIA's PC sampling approximates our needs	GPU vendors

• **System software must provide appropriate interfaces for introspection and analysis**
  • e.g. Linux perf_events supports sample-based performance monitoring even in the kernel
  • e.g. dynamic loader (ld.so) provides LD_AUDIT interface for monitoring and control of dynamic library operations
  • elfutils must support NVIDIA's extended line maps in CUDA 11.2+ GPU binaries
	Red Hat

• GPU vendor software stacks (kernel driver, runtime, tools API)	GPU vendors
• **Compiler must compute high-quality DWARF information** • associate each machine instruction with full call chains involving inlined templates and functions	Vendors and LLVM community
• **Runtime must maintain information needed to map computations to a source-level view** • OpenMP's OMPT helps bridge the vast gap between the implementation and user-level view	OpenMP Language Committee and LLVM Community

• **Performance tools must gather measurements using multiple modalities and map them to source**
  • precise attribution when possible
  • reconstruct approximate attribution when precise attribution is unavailable
  • GPU calling context
  • loops in CPU and GPU code
  • attribute inefficiencies from where they are observed back to their causes
	Wisconsin's Dyninst Project

Bonus Content

Download Hands-on Tutorial Examples on Ascent

- **git clone https://github.com/hpctoolkit/hpctoolkit-tutorial-examples**
- **Configured for data collection on ascent**
  - openmp example
    - miniqmc
      - CPU OpenMP: GCC, XL
  - gpu examples
    - quicksilver (dynamic monte-carol particle transport)
    - laghos (ultrashock code)
    - lammps (molecular dynamics)
    - pelec (compressible AMR combustion code)

Working with Hands-on Tutorial Examples on Ascent

- **Locate the examples configured for measurement on ascent**
  - from the directory hpctoolkit-tutorial-examples, you can run “find . -name ascent.sh” to see all of the examples prepared for ascent
  - change into one of the example directories, e.g. cd examples/gpu/quicksilver
  - source setup-env/ascent.sh
  - running make in a directory will show you the commands
  - typical commands
    - make build # build the example
    - make run # collect profiles and traces
    - make run-pc # collect detailed profiles using PC sampling
    - make view # view profile and trace results
    - make view-pc # view PC sampling results

Working with Hands-on Tutorial Examples on Theta-GPU

- **See theta:/grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit**

- **README with directions**
- **code-examples**
  - quicksilver.tgz - a dynamic monte carlo neutron transport miniapp
- **databases**
  - PeleC - 98 MPI ranks @ 5 GPU streams each (AMR turbulent combustion code)
  - hpcprof2-mpi - 8 MPI ranks @ 128 threads each (parallel analysis of 64K profiles)
  - Quicksilver - Single-threaded @ 1 GPU stream (dynamic Monte-Carlo neutron transport)

Measuring and Analyzing Quicksilver on Theta-GPU

- **Create a directory where you want to work**
- **Unpack the tar file with the example /grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit/code-examples/quicksilver.tgz**
- **Follow the directions in /grand/ATPESC2021/EXAMPLES/track-6-tools/hpctoolkit/code-examples/README**
- **Pro tip: use two windows**
  - one on a theta login node
  - one on a GPU compute node
- **Why:**
  - You can only compile and run the example on a GPU compute node
  - You can only run the GUI on a Theta login node
    - there is no X forwarding from GPU compute nodes

HPCToolkit's Graphical User Interfaces

- **Overview**
- **Tips for using them effectively**

hpctraceviewer Panes and their Purposes

- **Trace View pane**
  - Displays a sequence of samples for each trace line rendered
  - Title bar shows time interval rendered, rank interval rendered, cross hair location
- **Call Path pane**
  - Show the call path of the selected thread at the cross hair
- **Depth View pane**
  - Show the call stack over time for the thread marked by the cross hair
  - Unusual changes or clustering of deep call stacks can indicate behaviors of potential interest
- **Summary View pane**
  - At each point in time, a histogram of colors above in a vertical column of the Trace View

Rendering Traces with hpctraceviewer

- **hpctraceviewer renders traces by sampling the [rank x time] rectangle in the viewport**
  - Don’t try to summarize activity in a time interval represented by a pixel
  - Just pick the last activity before the sample point in time
- **Cost of rendering a large execution is [H x T lg N] for traces of length N**
  - The number of trace lines that can be rendered is limited by the number of vertical pixels H
  - Binary search along rendered trace lines to extract values for pixels
- **It can be used to analyze large data: thousands of ranks and threads**
  - Data is kept on disk, memory mapped, and read only as needed

Understanding How hpctraceviewer Paints Traces

- **CPU trace lines**
  - Given: (procedure f, t) (procedure g, t') (procedure h, t'')
    - Default painting algorithm
      - paint color "f" in [t,t'); paint color "g" in [t', t'')
    - Midpoint painting algorithm
      - paint color "f" in [t, (t+t')/2); paint color "g" in [(t+t')/2, (t'+t'')/2)
- **GPU trace lines**
  - Given GPU operations "f" in interval [t, t') and and "g" in interval [t'', t''')
    - paint color "f" in [t, t'); paint color white in [t', t''); paint color "g" in [t'', t''')

Analysis Strategies with Time-centric hpctraceviewer

- **Use top-down analysis to understand the broad characteristics of the parallel execution**
- **Click on a point of interest in the Trace View to see the call path there**
- **Zoom in on individual phases of the execution or more generally subsets of [rank, time]**
  - The mini-map tracks what subset of the execution you are viewing
- **Home, undo, redo buttons allow you to move back and forth in a sequence of zooms**
- **Drill down the call path to see what is going on at the call path leaves**
  - Hold your mouse over the call path depth selector. a tool tip will tell you the maximum depth
  - Type the maximum call stack depth number into the depth selector
- **Use the summary view to see a histogram about what fraction of threads or ranks is doing at each time**
- **The summary view can facilitate analysis of how behavior changes over time**
- **The statistics view can show you the fraction of [rank x time] spent in each procedure at the selected depth level**

Understanding the Navigation Pane in Code-centric hpcviewer

- **<program root>: the top of the call chain for the executable**
- **<thread root>: the top of the call chain for any pthreads**
- **<partial call paths>**
  - The presence of partial call paths indicates that hpcrun was unable to fully unwind the call stack
  - Even if a large fraction of call paths are “partial” unwinds, bottom-up and flat views can be very informative
- **Sometimes functions appear in the navigation pane and appear to be a root**
  - This means that hpcrun believed that the unwind was complete and successful
  - Ideally, this would have been placed under <partial call paths>

Understanding the Navigation Pane in Code-centric hpcviewer

- **Treat inlined functions as if regular functions**
- **Calling an inlined function**

[I] is a tag used to indicate that the called function is inlined

callsite is a hyperlink to the file and source line where the inlined function is called

callee is a hyperlink to the definition of the inlined function

- **If no source file is available, the caller line number and the callee will be in black**

Analysis Strategies with Code-centric hpcviewer

- **Use top-down analysis to understand the broad characteristics of the execution**
  - Are there specific unique subtrees in the computation that use or waste a lot of resources?
  - Select a costly node and drill down the “hottest path” rooted there with the flame button
  - One can select a node other than the root and use the flame button to look in its subtree
  - Hold your mouse over a long name in the navigation pane to see the full name in a tool tip
- **Use bottom-up analysis to identify costly procedures and their callers**
  - Pick a metric of interest, e.g. cycles
  - Sort by cycles in descending order
  - Pick the top routine and use the flame button to look up the call stack to its callers
  - Repeat for a few routines of particular interest, e.g. network wait, lock wait, memory alloc, …
- **Use the flat view to explore the full costs associated with code at various granularities**
  - Sort by a cost of interest; use the flame button to explore an interesting load module
  - Use the “flatten” button to melt away load modules, files, and functions to identify the most costly loop

Preparing a GPU-accelerated Program for HPCToolkit

- **HPCToolkit doesn't need any modifications to your Makefiles**
  - it can measure fully-optimized code without special preparation
- **To get the most from your measurement and analysis**
  - Compile your program with line numbers
    - CPU (all compilers)
      - add "-g" to your compiler optimization flags
    - NVIDIA GPUs
      - compiling with nvcc
        - add "-lineinfo" to your optimization flags for GPU line numbers
        - adding -G provides full information about inlining and GPU code structure but disables optimization
      - compiling with xlc
        - line information is unavailable for optimized code
    - AMD GPUs, no special preparation needed
      - current AMD GPUs and ROCM software stack lack capabilities for fine-grain measurement and attribution
    - Intel GPUs (prototypes not integrated into HPCToolkit master)
      - monitors kernel launches, memory copies, synchronization
      - partial support for fine-grain monitoring with GTPin instrumentation; no source-level attribution yet

Using HPCToolkit to Measure an Execution

- **Sequential program**
  - `hpcrun [measurement options] program [program args]`
- **Parallel program**
  - `mpirun -n <nodes> [mpi options] hpcrun [measurement options] \ program [program args]`
  - Similar launches with job managers
    - LSF: jsrun
    - SLURM: srun
    - Cray: aprun

CPU Time-based Sample Sources - Linux thread-centric timers

- **CPUTIME (DEFAULT if no sample source is specified)**
  - CPU time used by the thread in microseconds
  - Does not include time blocked in the kernel
    - **disadvantage: completely overlooks time a thread is blocked**
    - **advantage: a blocked thread is never unblocked by sampling**
- **REALTIME**
  - Real time used by the thread in microseconds
  - Includes time blocked in the kernel
    - **advantage: shows where a thread spends its time, even when blocked**
    - **disadvantages**
      - **activates a blocked thread to take a sample**
      - **a blocked thread appears active even when blocked**

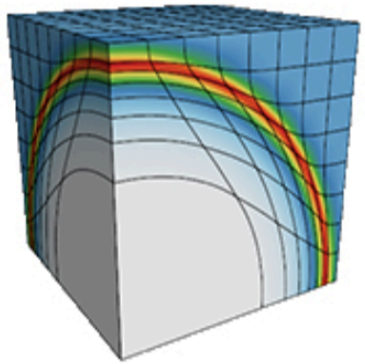
Note: Only use one Linux timer to measure an execution

CPU Sample Sources - Linux perf_event monitoring subsystem

- **Kernel subsystem for performance monitoring**
- **Access and manipulate**
  - Hardware counters: cycles, instructions, …
  - Software counters: context switches, page faults, …
- **Available in Linux kernels 2.6.31+**
- **Characteristics**
  - Monitors activity in user space and in the kernel
    - Can see costs in GPU drivers

Case Study: Measurement and Analysis of GPU-accelerated Laghos

Laghos (LAGrangian High-Order Solver) is a LLNL ASC co-design mini-app that was developed as part of the CEED software suite, a collection of software benchmarks, miniapps, libraries and APIs for efficient exascale discretization based on high-order finite element and spectral element methods.



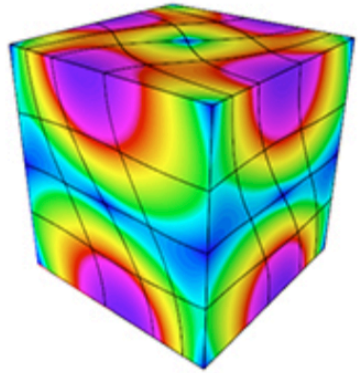


Figure credit: https://computing.llnl.gov/projects/co-design/laghos

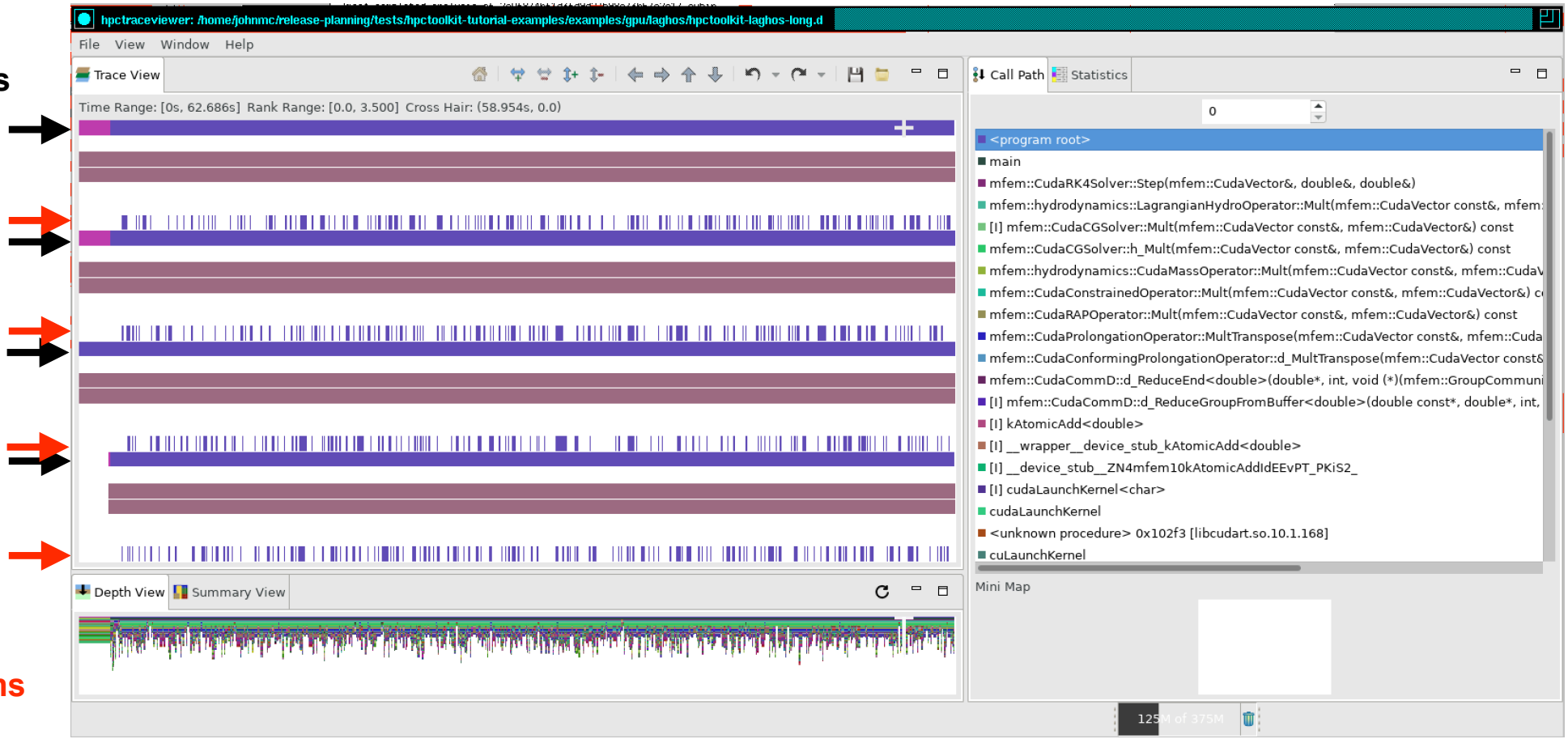
Applying the GPU Operation Measurement Workflow to Laghos

```
# measure an execution of laghos
time mpirun -np 4 hpcrun -o $OUT -e cycles -e gpu=nvidia -t \
   ${LAGHOS_DIR}/laghos -p 0 -m ${LAGHOS_DIR}/../data/square01_quad.mesh \
   -rs 3 -tf 0.75 -pa

# compute program structure information for the laghos binary
hpcstruct -j 16 laghos

# compute program structure information for the laghos cubins
hpcstruct -j 16 $OUT

# combine the measurements with the program structure information
mpirun -n 4  hpcprof-mpi -S laghos.hpcstruct $OUT
```

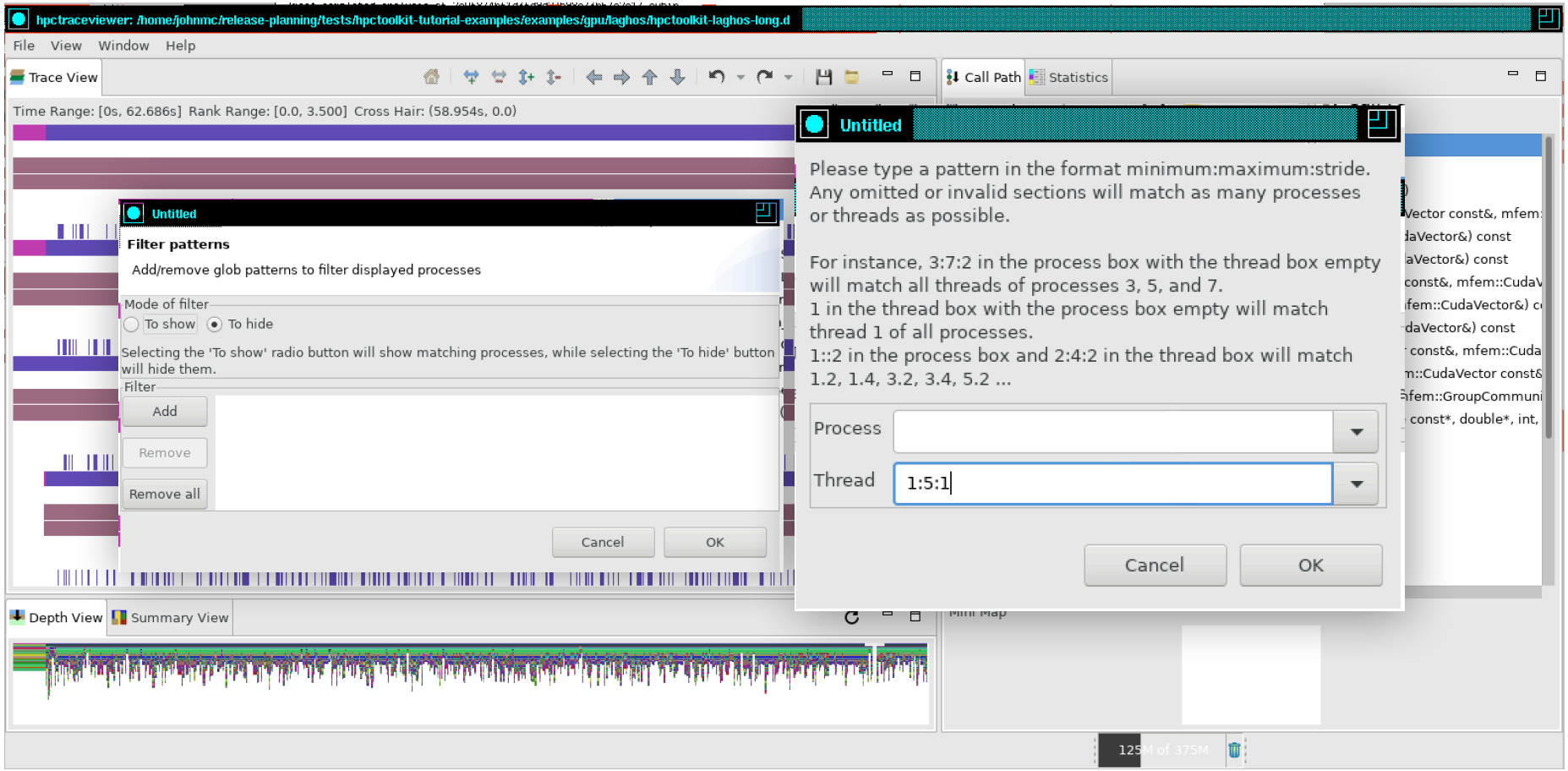
Computing Program Structure Information for NVIDIA cubins

- **When a GPU-accelerated application runs, HPCToolkit collects unique GPU binaries**
  - Currently, NVIDIA does not provide an API that provides a URI for cubins it launches
  - CUPTI presents cubins to tools as an interval in the heap (starting address, length)
  - HPCToolkit computes an MD5 hash for each cubin and saves one copy
    - stores save cubins in hpcrun's measurement directory: <measurement directory>/cubins
- **Analyze the cubins collected during an execution**
  - **`hpcstruct -j 16 <measurement directory>`**
    - lightweight analysis based only on cubin symbols and line map
  - **`hpcstruct -j 16 –gpucfg yes <measurement directory>`**
    - heavyweight analysis based only on cubin symbols, line map, control flow graph
      - uses nvdisasm to compute control flow graph
    - fine-grain analysis only needed to interpret PC sampling experiments
  - **`hpcstruct`** analyzes cubins in parallel using thread count specified with -j

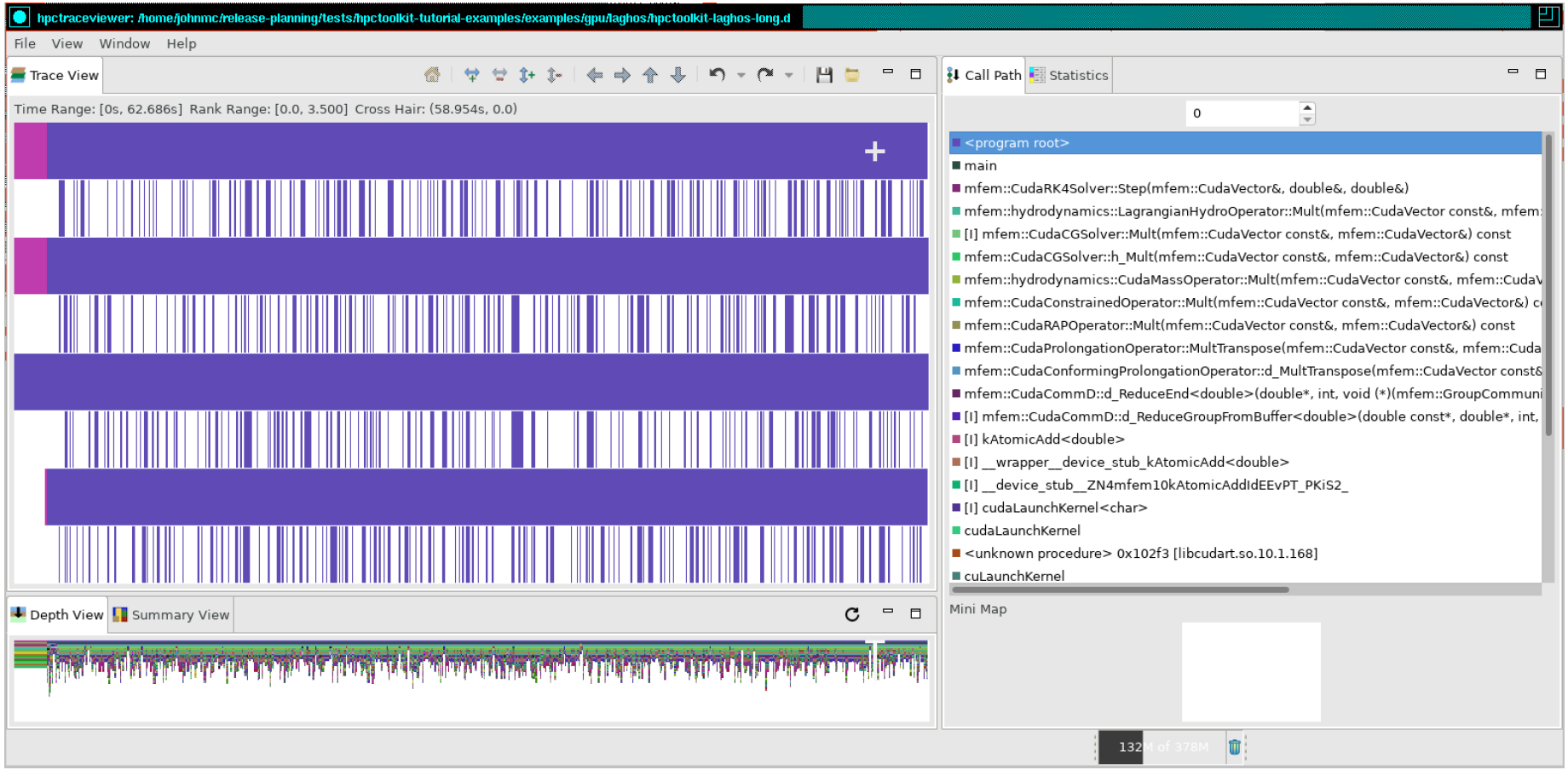
Initial hpctraceviewer view of Laghos (long) Execution



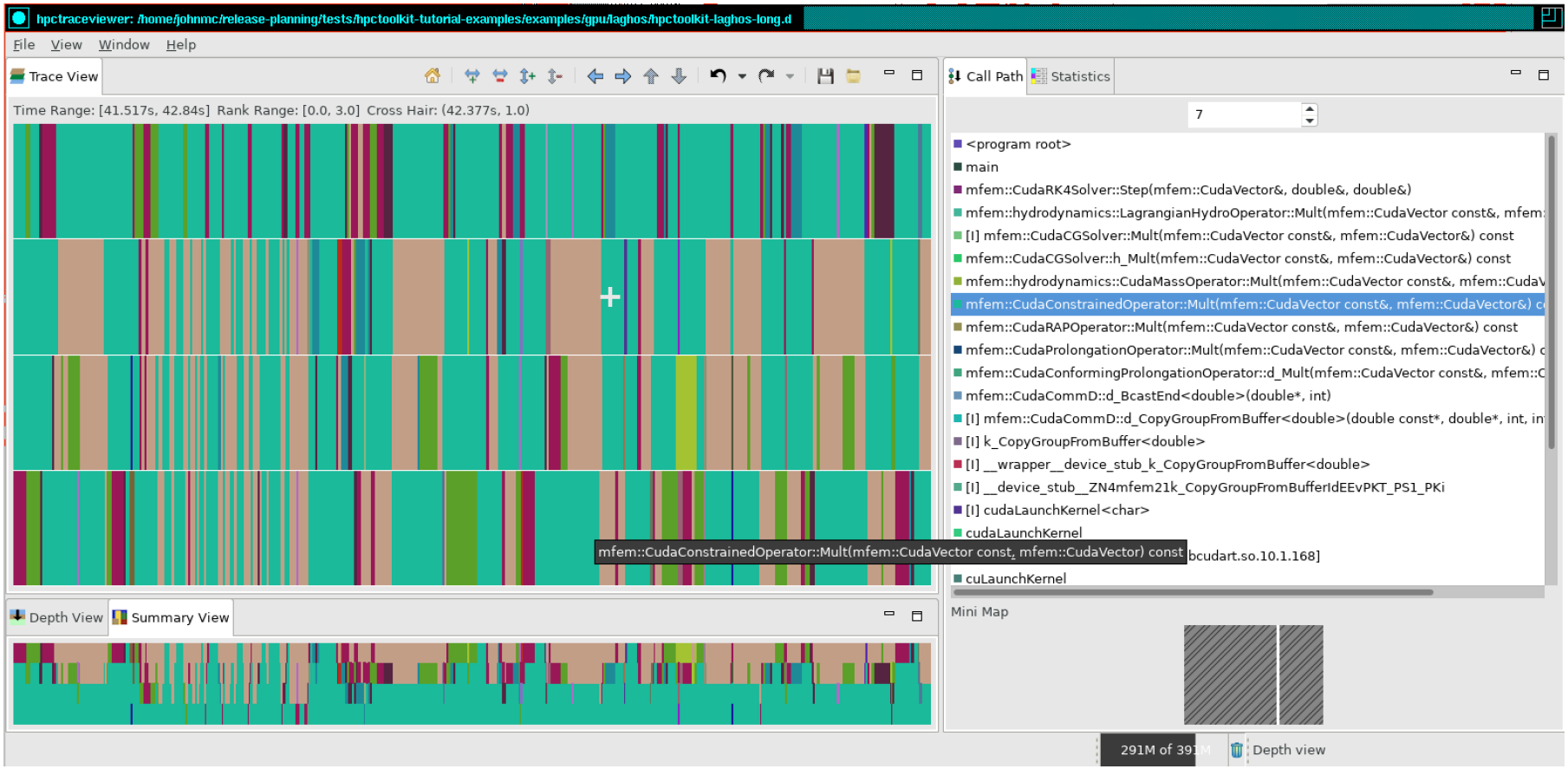
Hiding the Empty MPI Helper Threads



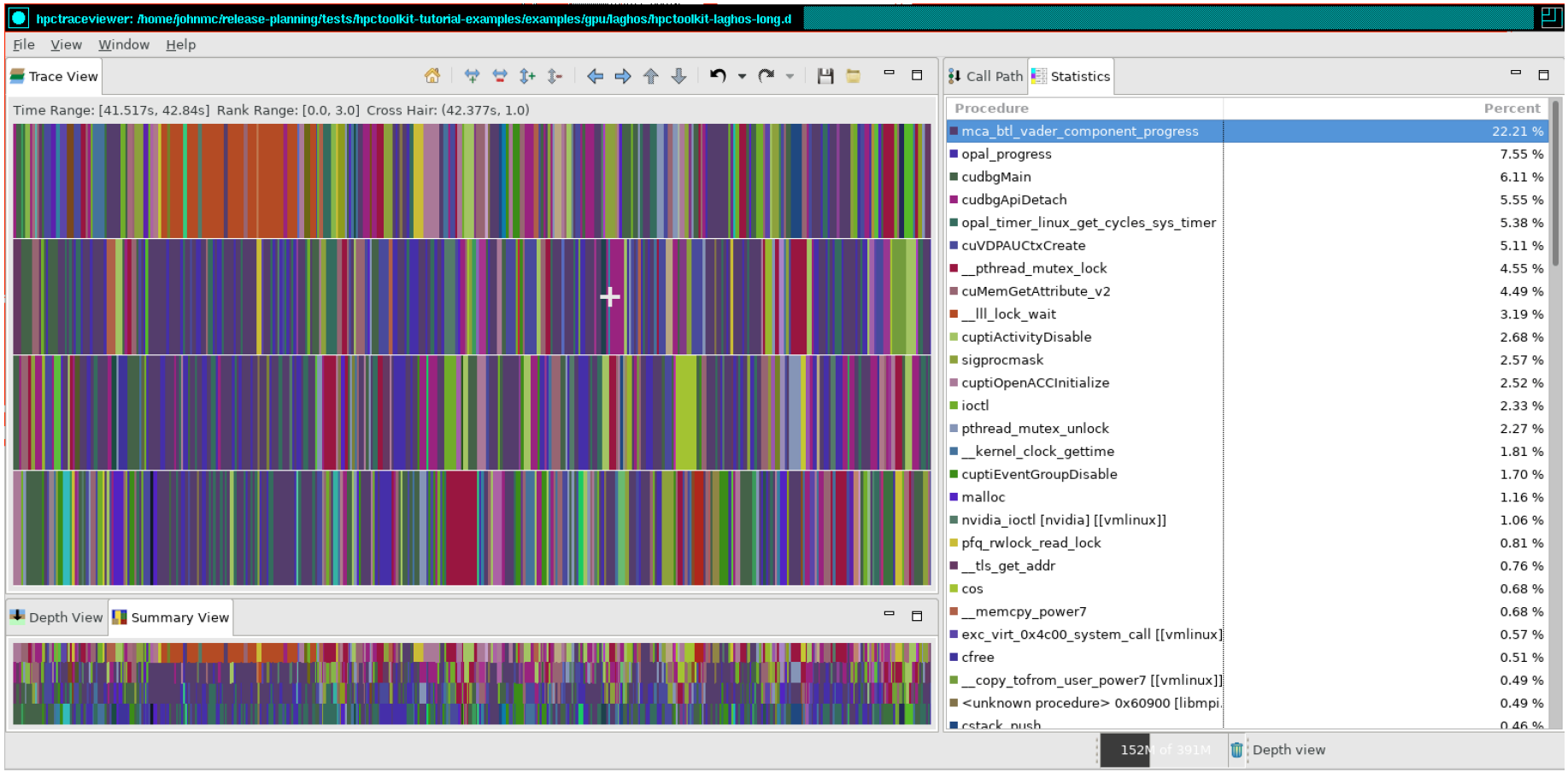
After Hiding the Empty MPI Helper Threads



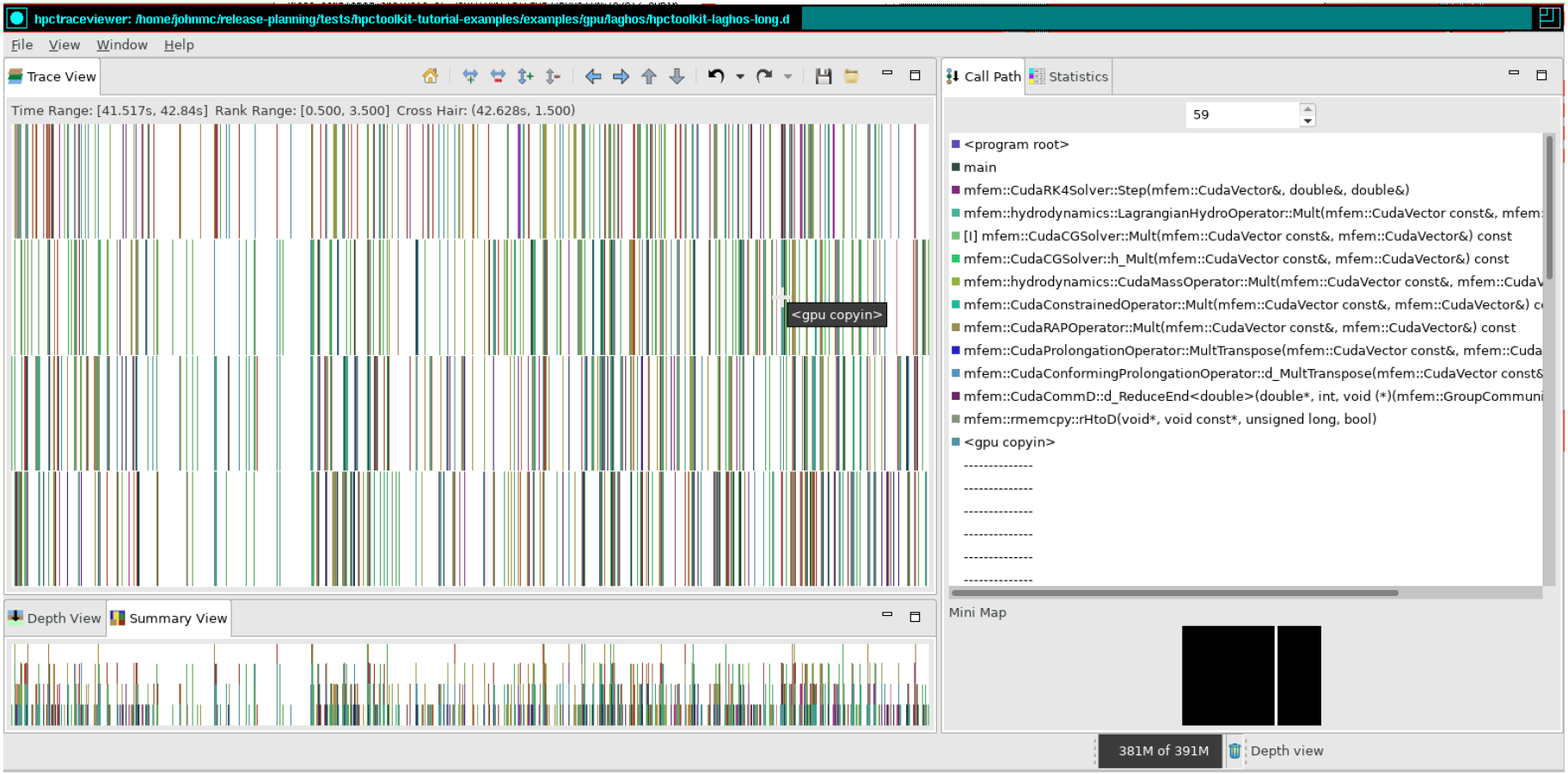
A Detail of Only the MPI Threads



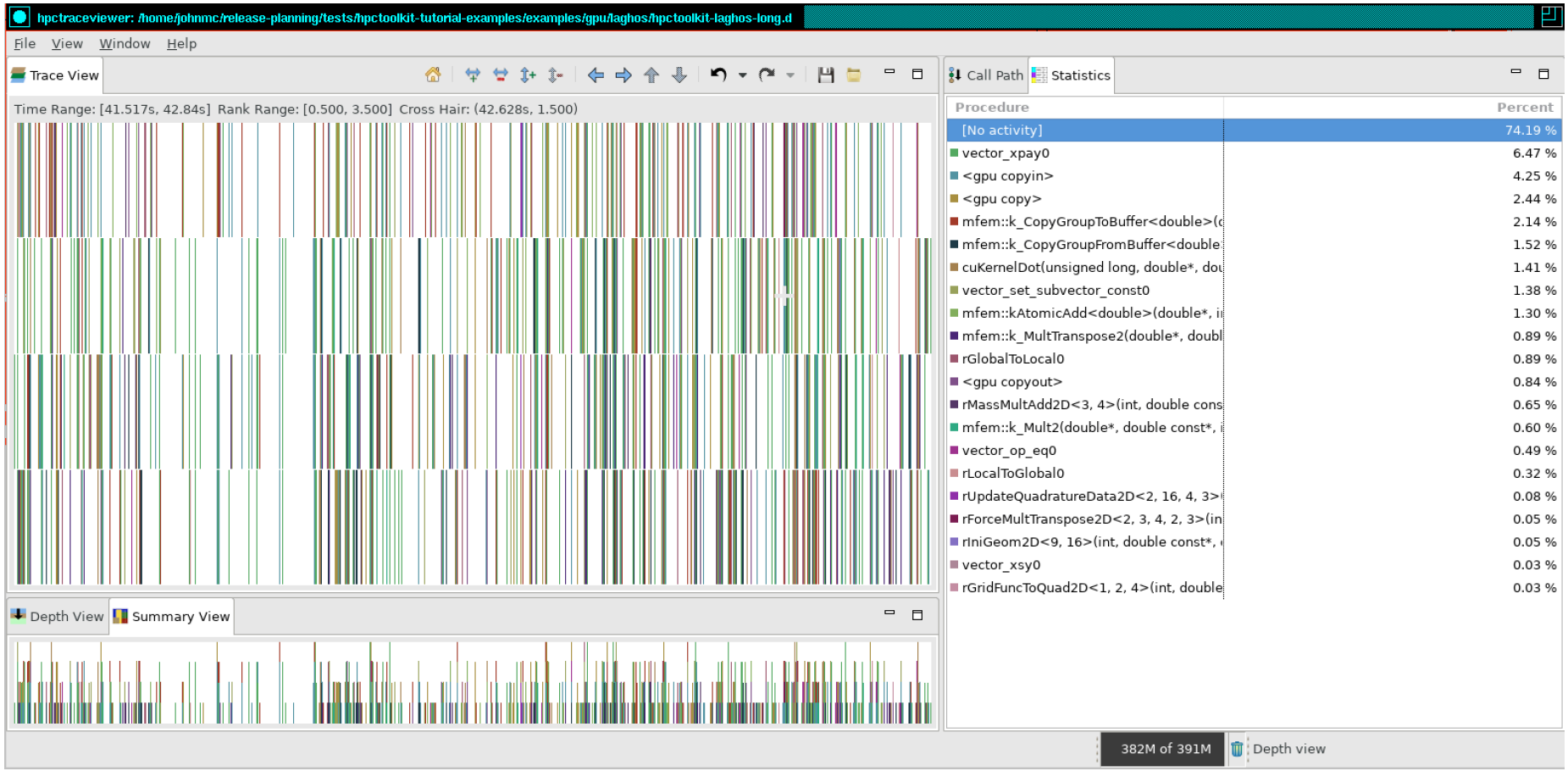
Only the MPI Threads - Analysis using the Statistics Panel



Only the GPU Threads - Inspecting the Callpath for a Kernel



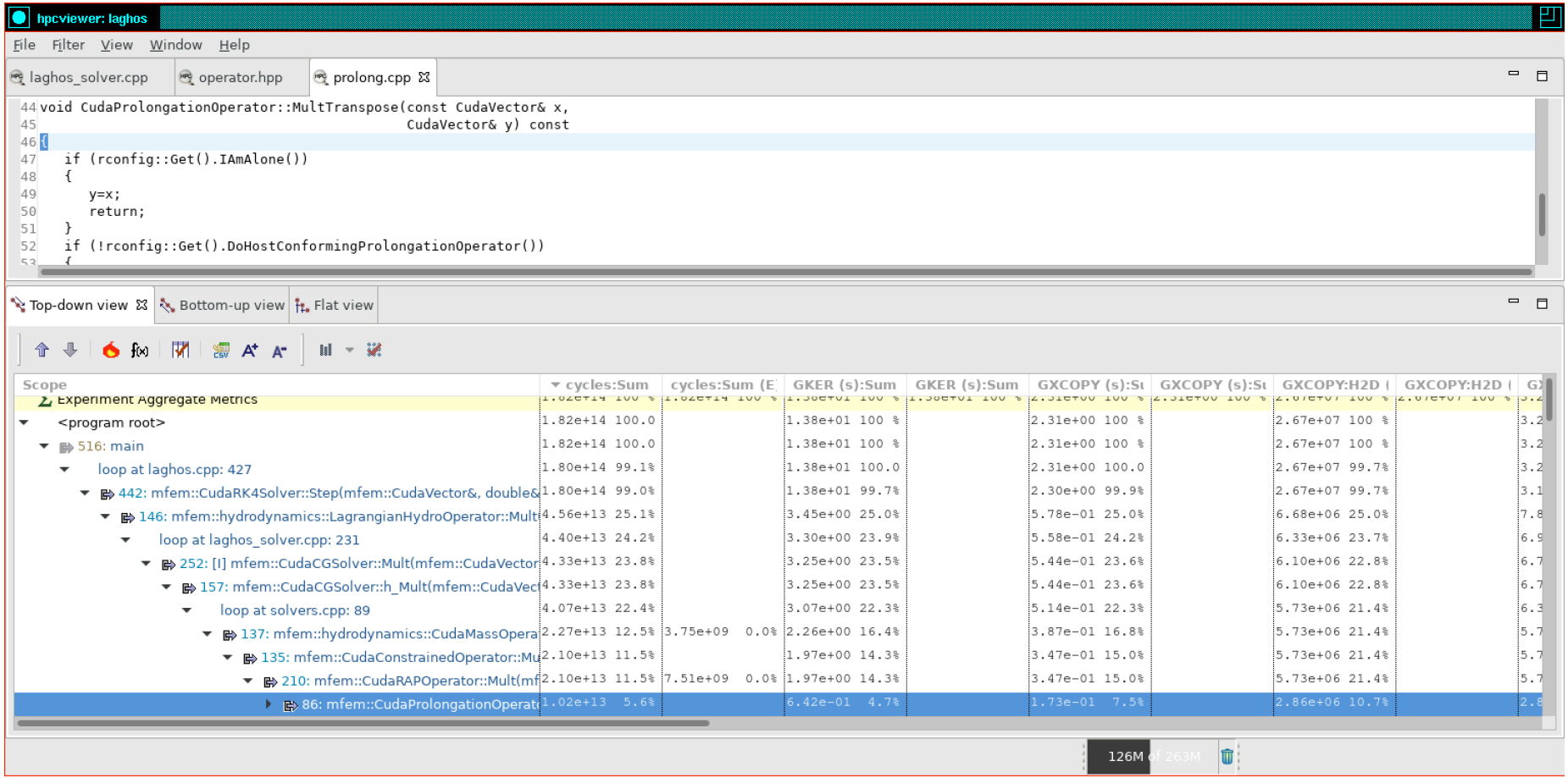
Only the GPU Threads - Analysis Using the Statistics Panel



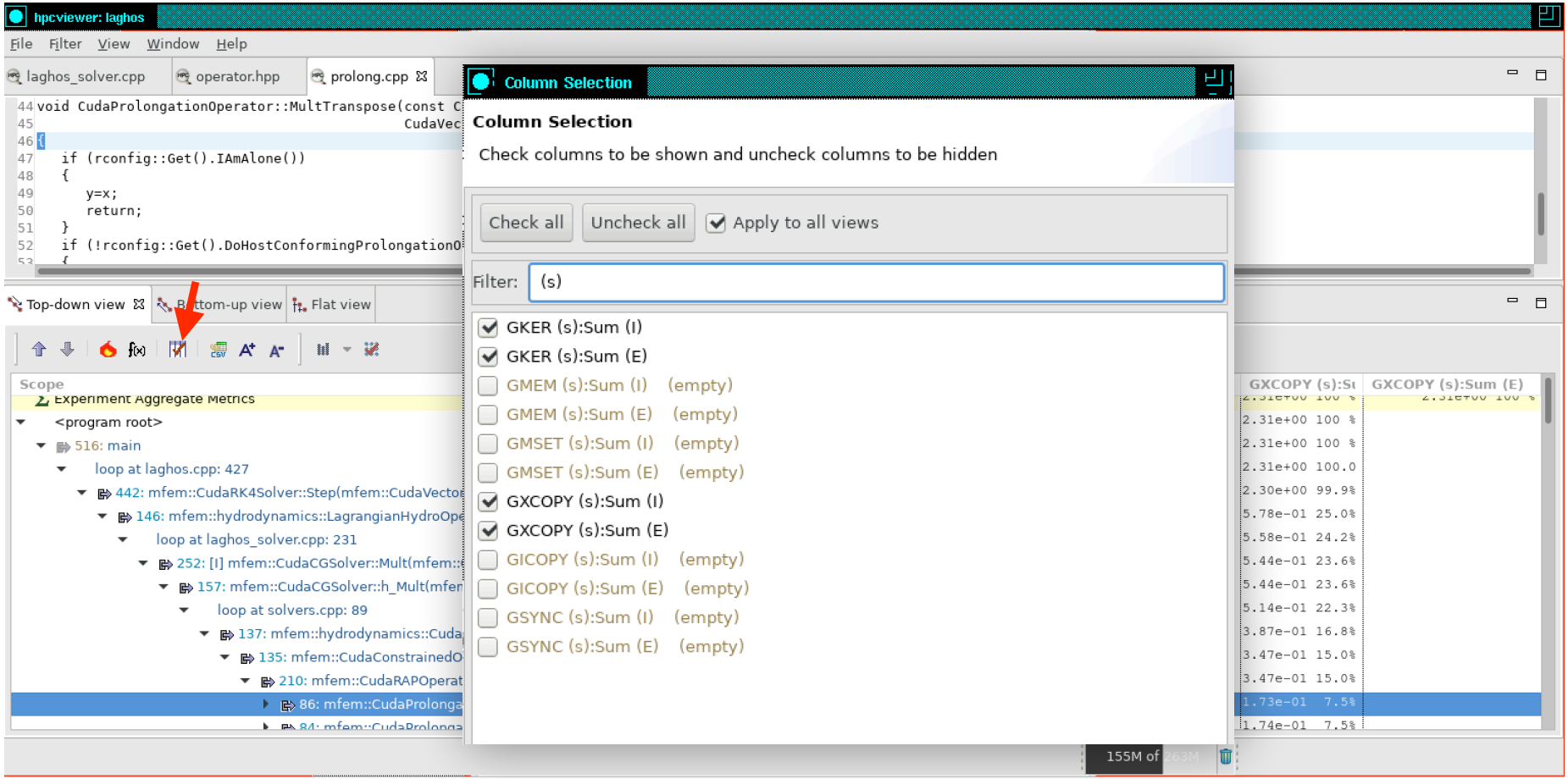
Some Cautions When Analyzing GPU Traces

- **There are overheads introduced by NVIDIA's monitoring API that we can't avoid**
- **When analyzing traces from your program and compare GPU activity to [no activity]**
  - Time your program without any tools
  - Time your program when tracing with HPCToolkit or nvprof
  - Re-weight <no activity> by the ratio of unmonitored time to monitored time
- **While this is a concern for traces, this should be less a concern for profiles**
  - On the CPU, HPCToolkit compensates for monitoring overhead in profiles by not measuring it

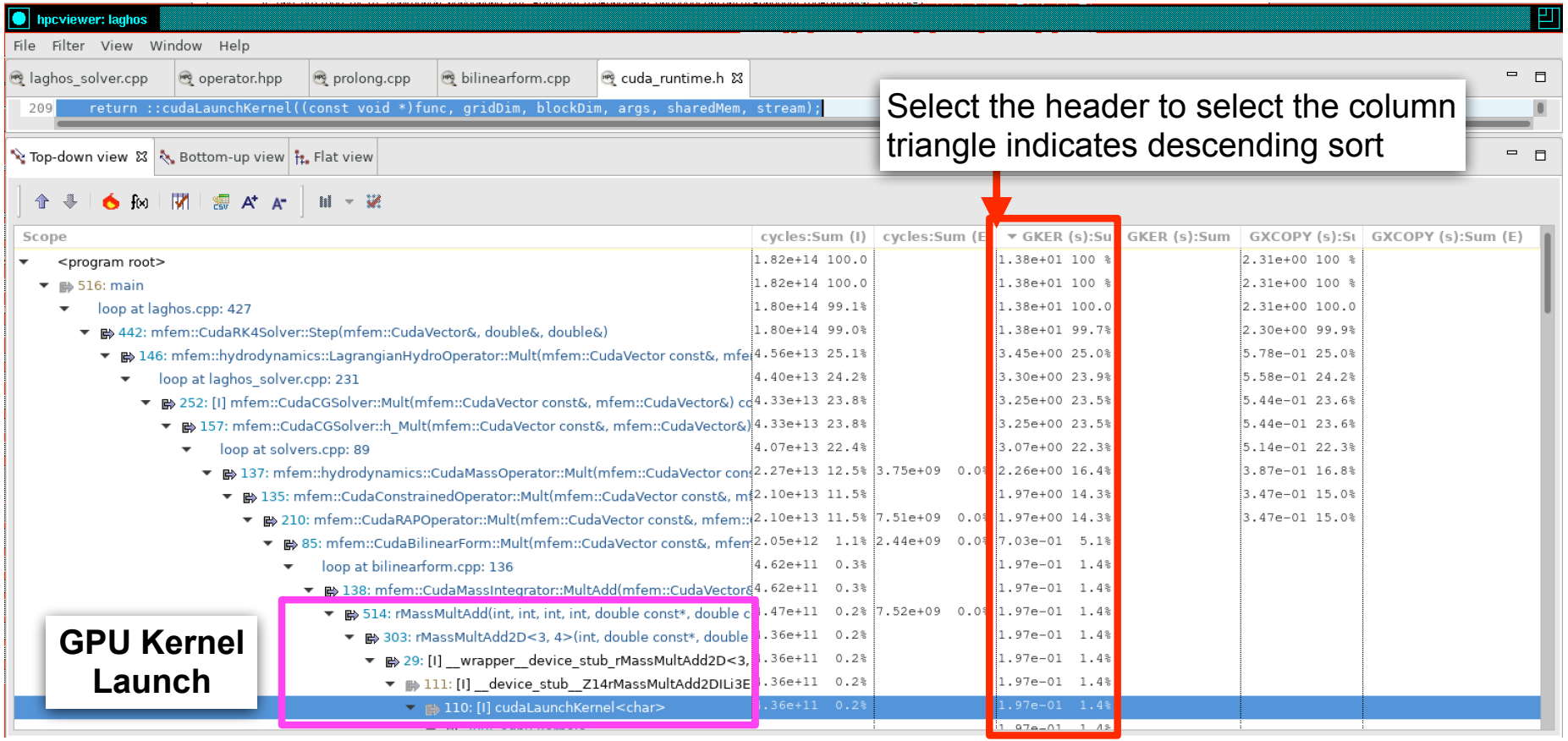
Using hpcviewer to See the Source-centric View



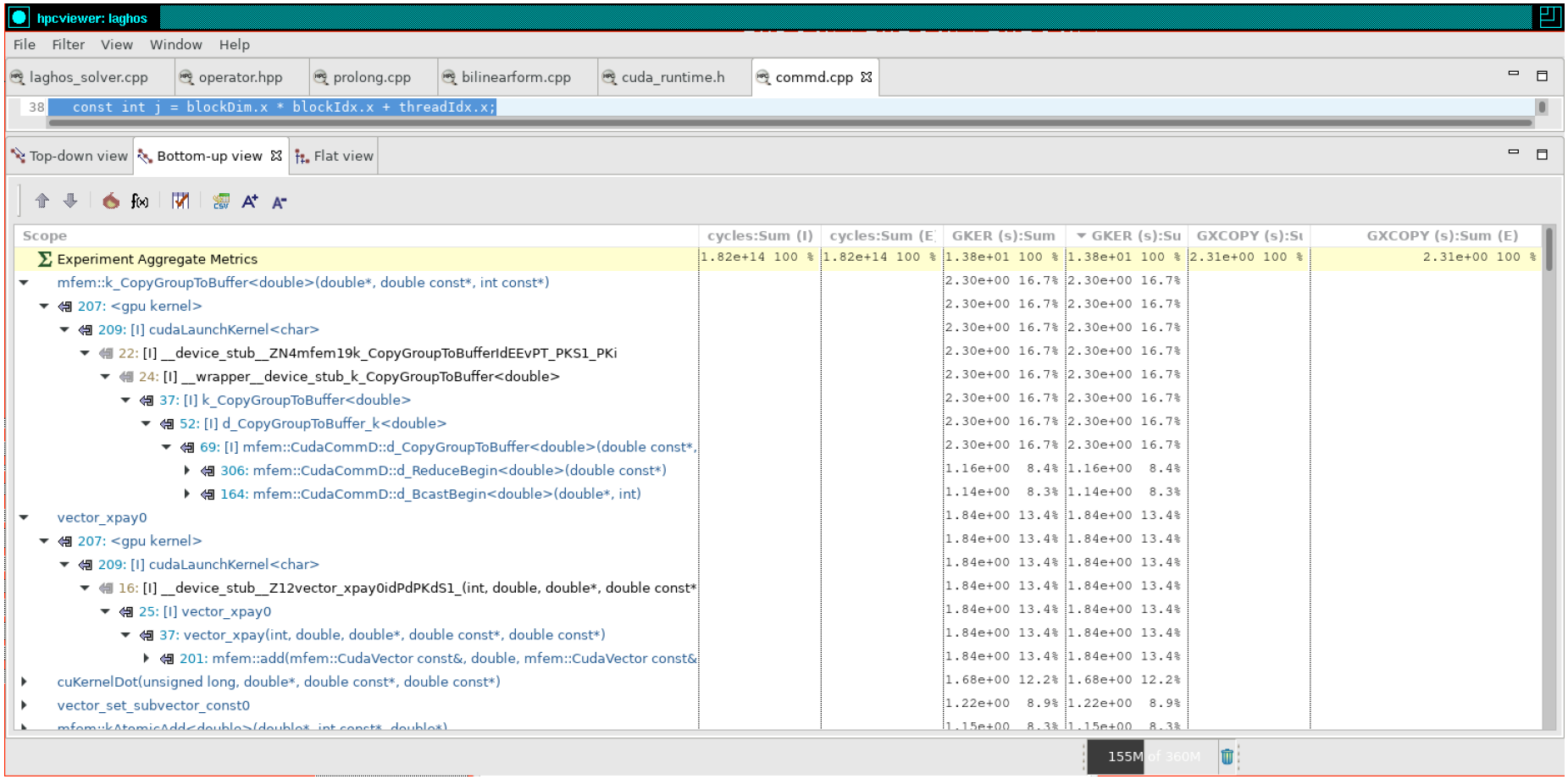
Selecting Metrics to Display Using the Column Selector



Using GPU Kernel Time to Guide Top-down Exploration



Using GPU Kernel Time to Guide Bottom-up Exploration



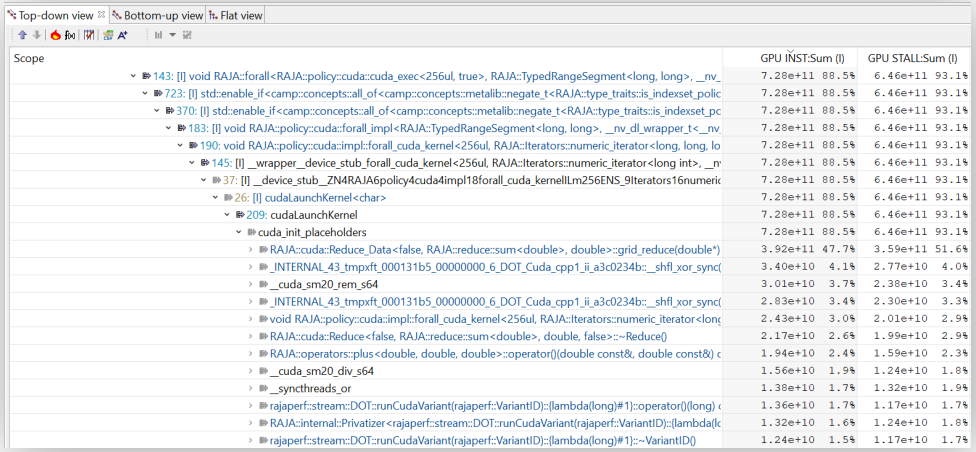
HPCToolkit's GPU Instruction Sampling Metrics (NVIDIA Only)

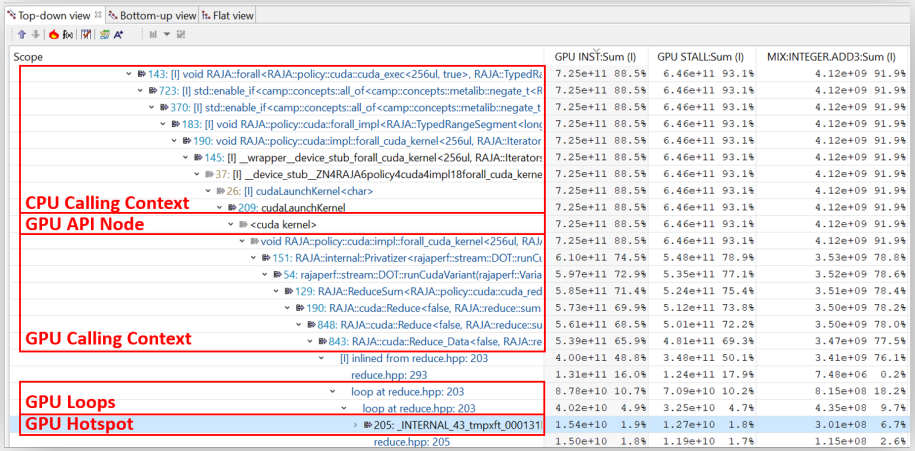
Metric	Definition
GINST:STL_ANY	GPU instruction stalls: any (sum of all **STALL** metrics other than **NONE**)
GINST:STL_NONE	GPU instruction stalls: no stall
GINST:STL_IFET	GPU instruction stalls: await availability of next instruction (fetch or branch delay)
GINST:STL_IDEP	GPU instruction stalls: await satisfaction of instruction input dependence
GINST:STL_GMEM	GPU instruction stalls: await completion of global memory access
GINST:STL_TMEM	GPU instruction stalls: texture memory request queue full
GINST:STL_SYNC	GPU instruction stalls: await completion of thread or memory synchronization
GINST:STL_CMEM	GPU instruction stalls: await completion of constant or immediate memory access
GINST:STL_PIPE	GPU instruction stalls: await completion of required compute resources
GINST:STL_MTHR	GPU instruction stalls: global memory request queue full
GINST:STL_NSEL	GPU instruction stalls: not selected for issue but ready
GINST:STL_OTHR	GPU instruction stalls: other
GINST:STL_SLP	GPU instruction stalls: sleep

Approximation of GPU Calling Contexts to Understand Performance

- GPU code from C++ template-based programming models is complex
- NVIDIA GPUs collect flat PC samples
- Flat profiles for instantiations of complex C++ templates are inscrutable

- HPCToolkit reconstructs approximate GPU calling contexts
  - Reconstruct call graph from machine code
  - Infer calls at call sites
    - PC samples of call instructions indicate calls
      - Use call counts to apportion costs to call sites
    - PC samples in a routine





Approximation of GPU Calling Contexts to Understand Performance

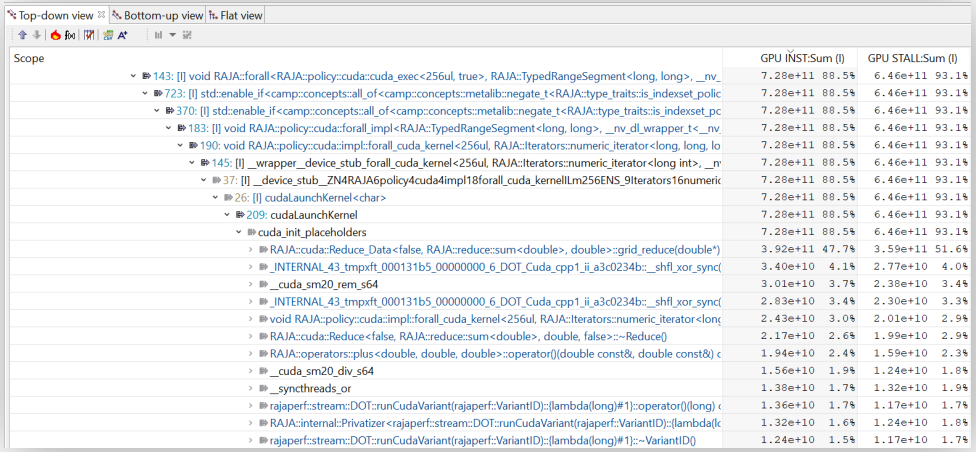
Top-down view | Bottom-up view | Flat view

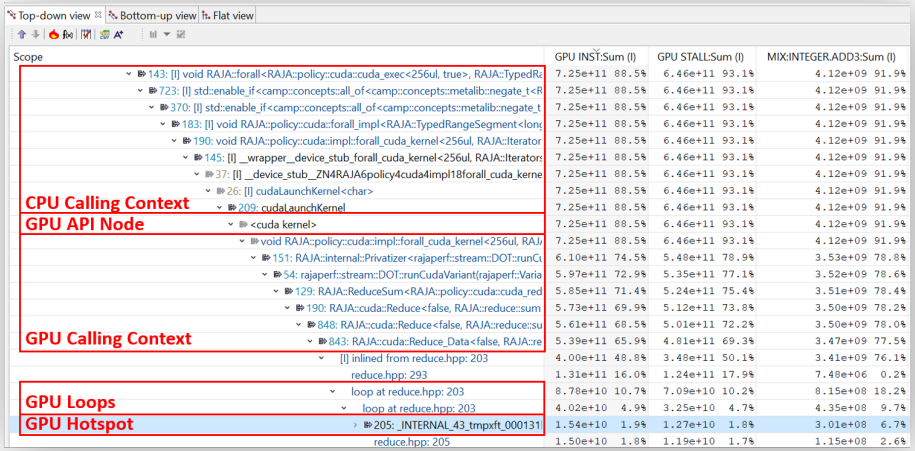
Scope	GPU INST:Sum (I)	GPU STALL:Sum (I)
143: [I] void RAJA::forall<RAJA::policy::cuda::cuda_exec<256ul, true>, RAJA::TypedRangeSegment<long, long>, __nv_	7.28e+11 88.5%	6.46e+11 93.1%
723: [I] std::enable_if<camp::concepts::all_of<camp::concepts::metalib::negate_t<RAJA::type_traits::is_indexset_polic	7.28e+11 88.5%	6.46e+11 93.1%
370: [I] std::enable_if<camp::concepts::all_of<camp::concepts::metalib::negate_t<RAJA::type_traits::is_indexset_pc	7.28e+11 88.5%	6.46e+11 93.1%
183: [I] void RAJA::policy::cuda::forall_impl<RAJA::TypedRangeSegment<long, long>, __nv_dl_wrapper_t<__nv_	7.28e+11 88.5%	6.46e+11 93.1%
190: void RAJA::policy::cuda::impl::forall_cuda_kernel<256ul, RAJA::Iterators::numeric_iterator<long, long, lo	7.28e+11 88.5%	6.46e+11 93.1%
145: [I] __wrapper__device_stub_forall_cuda_kernel<256ul, RAJA::Iterators::numeric_iterator<long int>, __n	7.28e+11 88.5%	6.46e+11 93.1%
37: [I] __device_stub__ZN4RAJA6policy4cuda4impl18forall_cuda_kernelILm256ENS_9Iterators16numeric	7.28e+11 88.5%	6.46e+11 93.1%
26: [I] cudaLaunchKernel<char>	7.28e+11 88.5%	6.46e+11 93.1%
209: cudaLaunchKernel	7.28e+11 88.5%	6.46e+11 93.1%
cuda_init_placeholders	7.28e+11 88.5%	6.46e+11 93.1%
RAJA::cuda::Reduce_Data<false, RAJA::reduce::sum<double>, double>::grid_reduce(double*)	3.92e+11 47.7%	3.59e+11 51.6%
_INTERNAL_43_tmpxft_000131b5_00000000_6_DOT_Cuda_cpp1_ii_a3c0234b::__shfl_xor_sync(	3.40e+10 4.1%	2.77e+10 4.0%
__cuda_sm20_rem_s64	3.01e+10 3.7%	2.38e+10 3.4%
_INTERNAL_43_tmpxft_000131b5_00000000_6_DOT_Cuda_cpp1_ii_a3c0234b::__shfl_xor_sync(	2.83e+10 3.4%	2.30e+10 3.3%
void RAJA::policy::cuda::impl::forall_cuda_kernel<256ul, RAJA::Iterators::numeric_iterator<long	2.43e+10 3.0%	2.01e+10 2.9%
RAJA::cuda::Reduce<false, RAJA::reduce::sum<double>, double, false>::~Reduce()	2.17e+10 2.6%	1.99e+10 2.9%
RAJA::operators::plus<double, double, double>::operator()(double const&, double const&) c	1.94e+10 2.4%	1.59e+10 2.3%
__cuda_sm20_div_s64	1.56e+10 1.9%	1.24e+10 1.8%
__syncthreads_or	1.38e+10 1.7%	1.32e+10 1.9%
rajaperf::stream::DOT::runCudaVariant(rajaperf::VariantID)::{lambda(long)#1}::operator()(long) c	1.36e+10 1.7%	1.17e+10 1.7%
RAJA::internal::Privatizer<rajaperf::stream::DOT::runCudaVariant(rajaperf::VariantID)::{lambda(lc	1.32e+10 1.6%	1.24e+10 1.8%
rajaperf::stream::DOT::runCudaVariant(rajaperf::VariantID)::{lambda(long)#1}::~VariantID()	1.24e+10 1.5%	1.17e+10 1.7%

Approximation of GPU Calling Contexts to Understand Performance

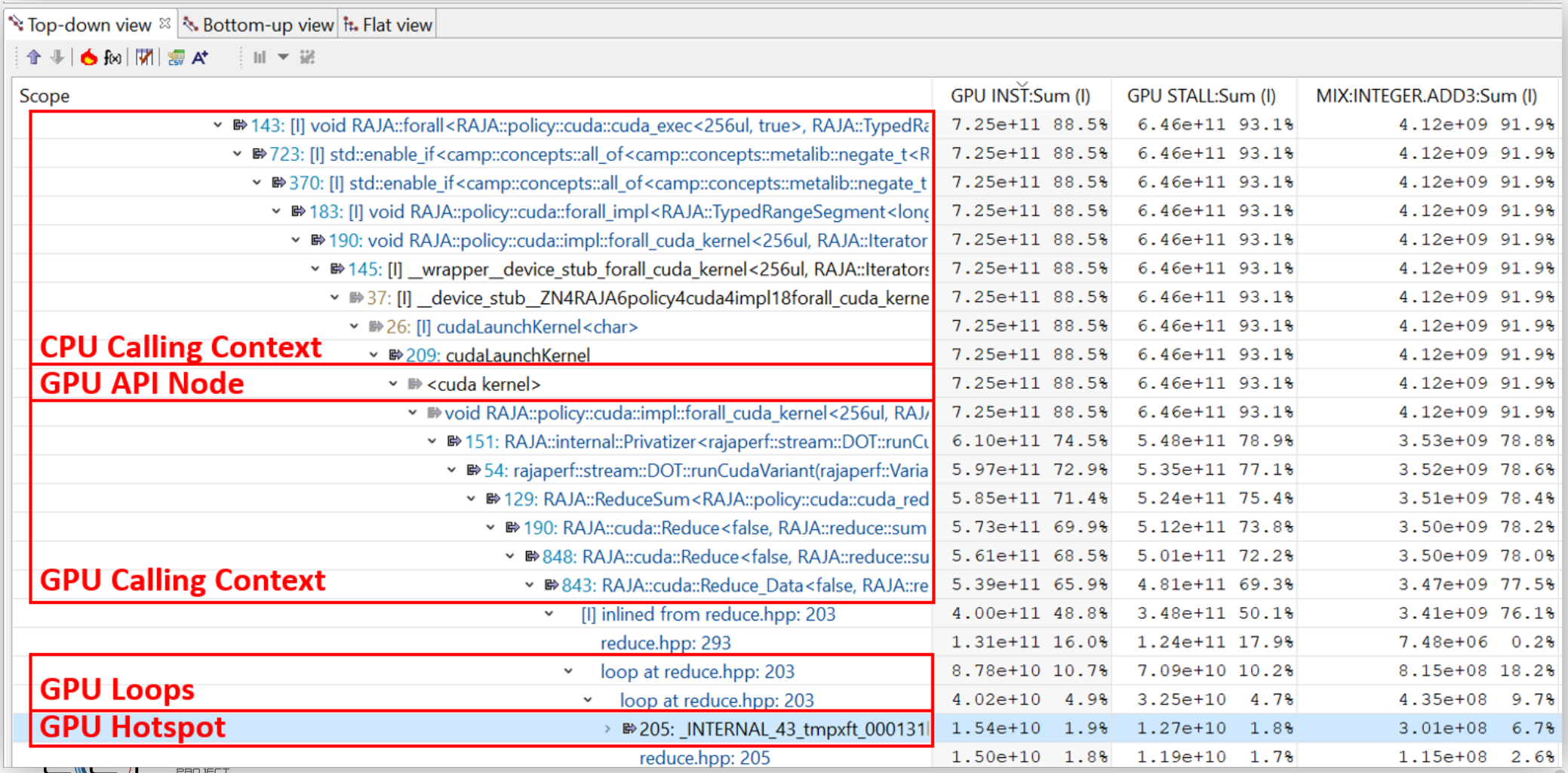
- GPU code from C++ template-based programming models is complex
- NVIDIA GPUs collect flat PC samples
- Flat profiles for instantiations of complex C++ templates are inscrutable

- HPCToolkit reconstructs approximate GPU calling contexts
  - Reconstruct call graph from machine code
  - Infer calls at call sites
    - PC samples of call instructions indicate calls
      - Use call counts to apportion costs to call sites
    - PC samples in a routine





Approximation of GPU Calling Contexts to Understand Performance



Accuracy of GPU Calling Context Recovery: Case Studies

- **Compute approximate call counts as the basis for partitioning the cost of function invocations across call sites**
  - Use call samples at call sites, data flow analysis to propagate call approximation upward
    - if samples were collected in some function f, if no calls to f were sampled, equally attribute f to each of its call sites
  - How accurate is our approximation?
- **Evaluation methodology**
  - Use NVIDIA's nvbit to
    - instrument call and return for GPU functions
    - instrument basic blocks to collect block histogram

Accuracy of GPU Calling Context Recovery: Case Studies

- **Error partitioning a function's cost among call sites**

$$Error = \sqrt{\sum_{i=0}^{n-1} \frac{\left(\sqrt{\sum_{j=0}^{i_c-1} \frac{(f_N(i,j)-f_H(i,j))^2}{i_c}}\right)^2}{n}}$$

geometric mean across GPU functions of (root mean square error of call attribution across all of a function's call sites comparing our approximation vs. attribution using exact nvbit measurements)

- **Experimental study**

Test Case	Unique Call Paths	Error
Basic_INIT_VIEW1D_OFFSET	9	0
Basic_REDUCE3_INT	113	0.03
Stream_DOT	60	0.006
Stream_TRIAD	5	0
Apps_PRESSURE	6	0
Apps_FIR	5	0
Apps_DEL_DOT_VEC_2D	3	0
Apps_VOL3D	4	0

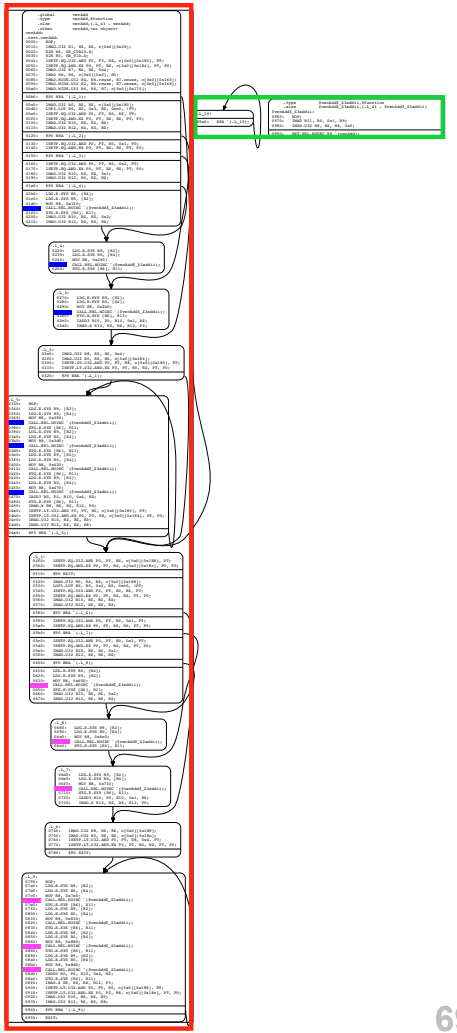
Costs of GPU Functions Distributed Among Their Call Sites

- **Use call site frequency approximation**
- **Use Gprof assumption: all calls to a function incur exactly the same cost**
  - known to not be true in all cases, but a useful assumption nevertheless

GPU call site attribution example

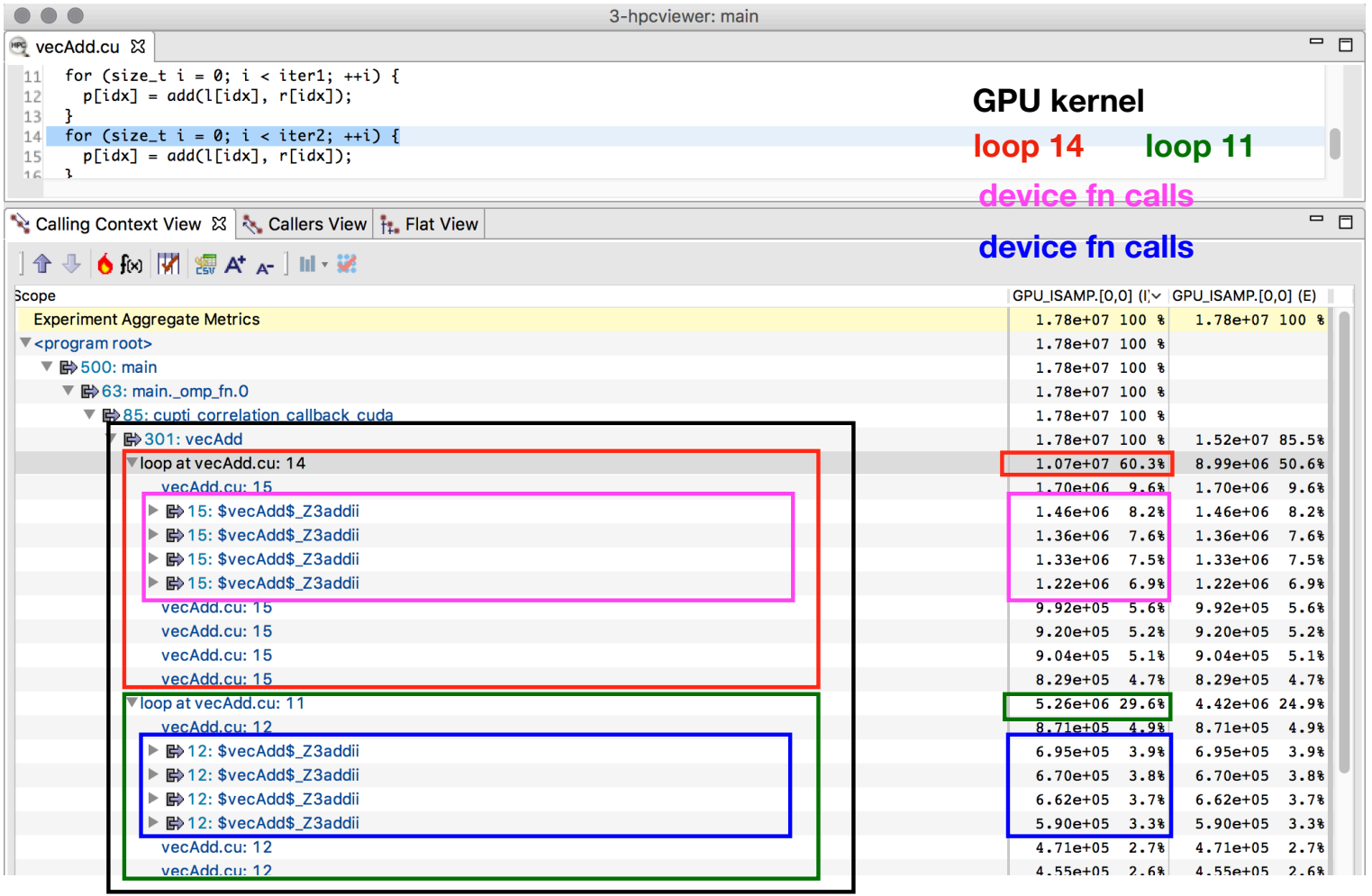
- **Case study: call function GPU “vectorAdd”***
  - iter1 = N
  - iter2 = 2N

```
1  __device__
2  int __attribute__ ((noinline)) add(int a, int b) {
3    return a + b;
4  }
5
6
7  extern "C"
8  __global__
9  void vecAdd(int *l, int *r, int *p, size_t N, size_t iter1,
       size_t iter2) {
10   size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
11   for (size_t i = 0; i < iter1; ++i) {
12     p[idx] = add(l[idx], r[idx]);
13   }
14   for (size_t i = 0; i < iter2; ++i) {
15     p[idx] = add(l[idx], r[idx]);
16   }
17 }
18
```



Note: the computation by the function is synthetic and is not a vector addition. The name came from code that was hacked to do perform an unrelated computation.

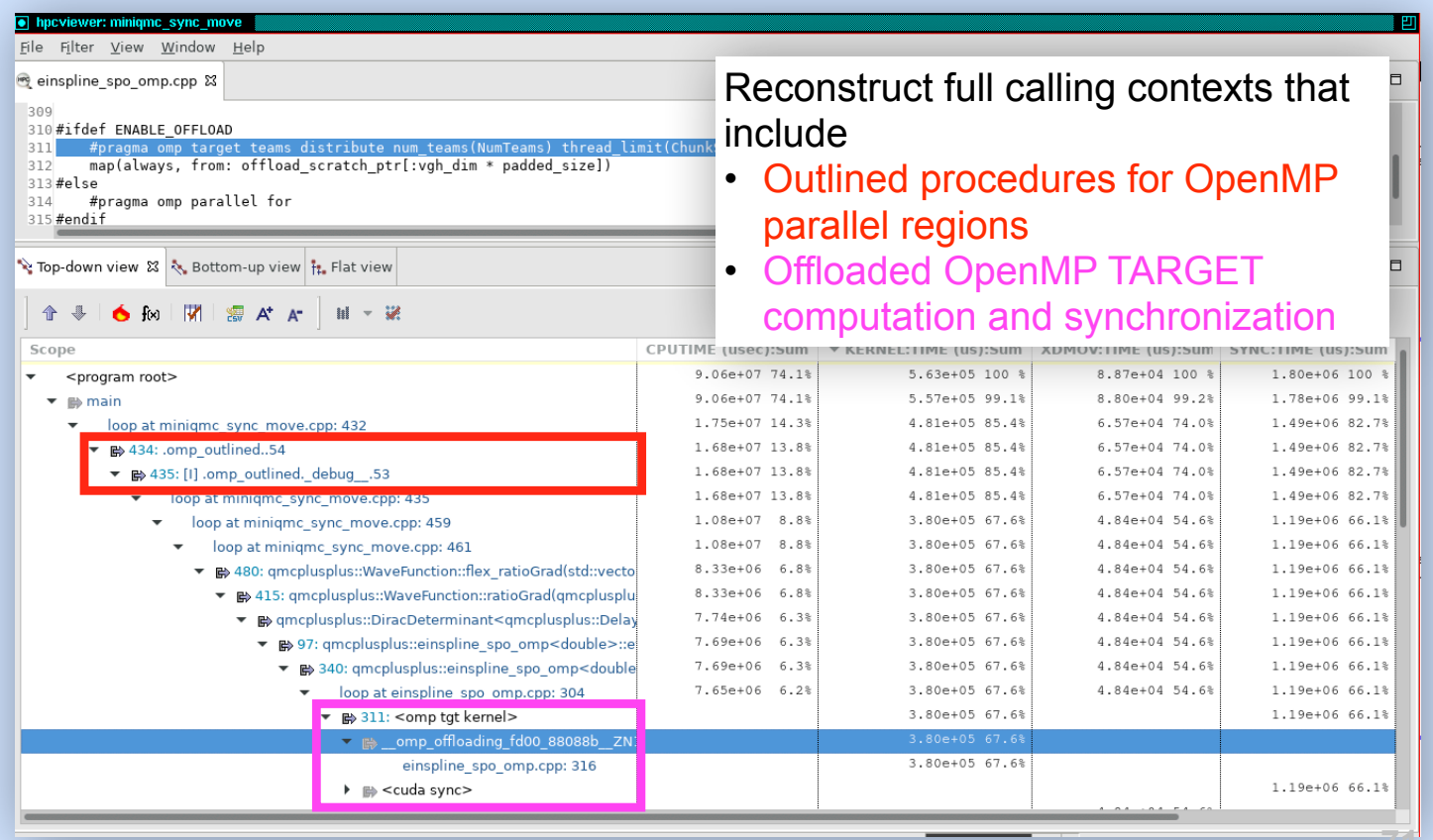
Profiling Result for GPU-accelerated Example



Support for OpenMP TARGET

- HPCToolkit implementation of OMPT OpenMP API
  - host monitoring
    - leverages callbacks for regions, threads, tasks
    - employs OMPT API for call stack introspection
  - GPU monitoring
    - leverages callbacks for device initialization, kernel launch, data operations
  - reconstruction of user-level calling contexts
- Leverages implementation of OMPT in LLVM OpenMP and libomptarget

ECP QMCPACK Project: miniqmc using OpenMP TARGET (Power9 + NVIDIA V100)



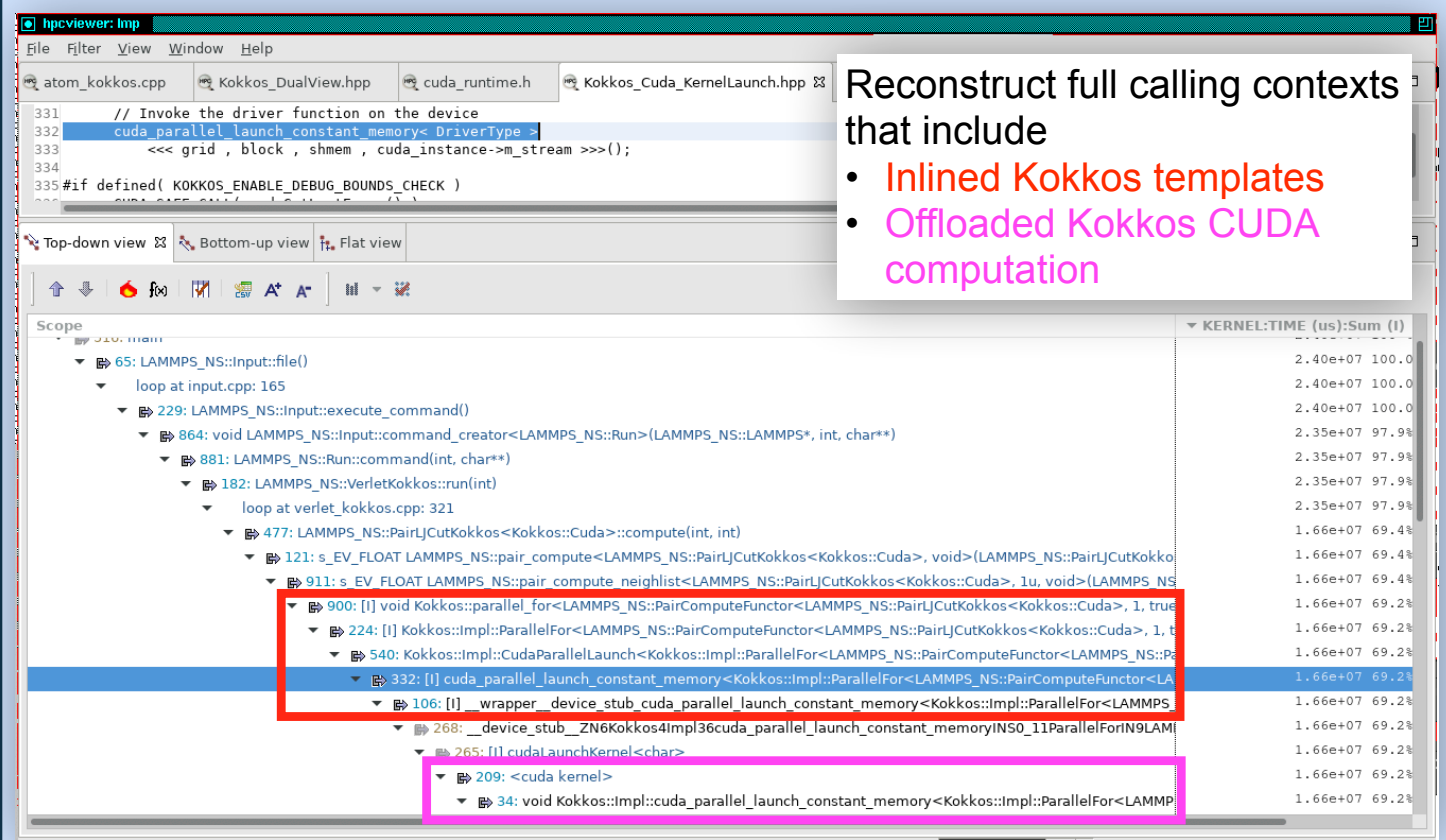
Reconstruct full calling contexts that include

- Outlined procedures for OpenMP parallel regions
- Offloaded OpenMP TARGET computation and synchronization

Support for RAJA and and Kokkos C++ Template-based Models

- RAJA and Kokkos provide portability layers atop C++ template-based programming abstractions
- HPCToolkit employs binary analysis to recover information about procedures, inlined functions and templates, and loops
  - Enables both developers and users to understand complex template instantiation present with these models

ECP EXAALT Project: lammps using Kokkos over CUDA (Power9 + NVIDIA V100)



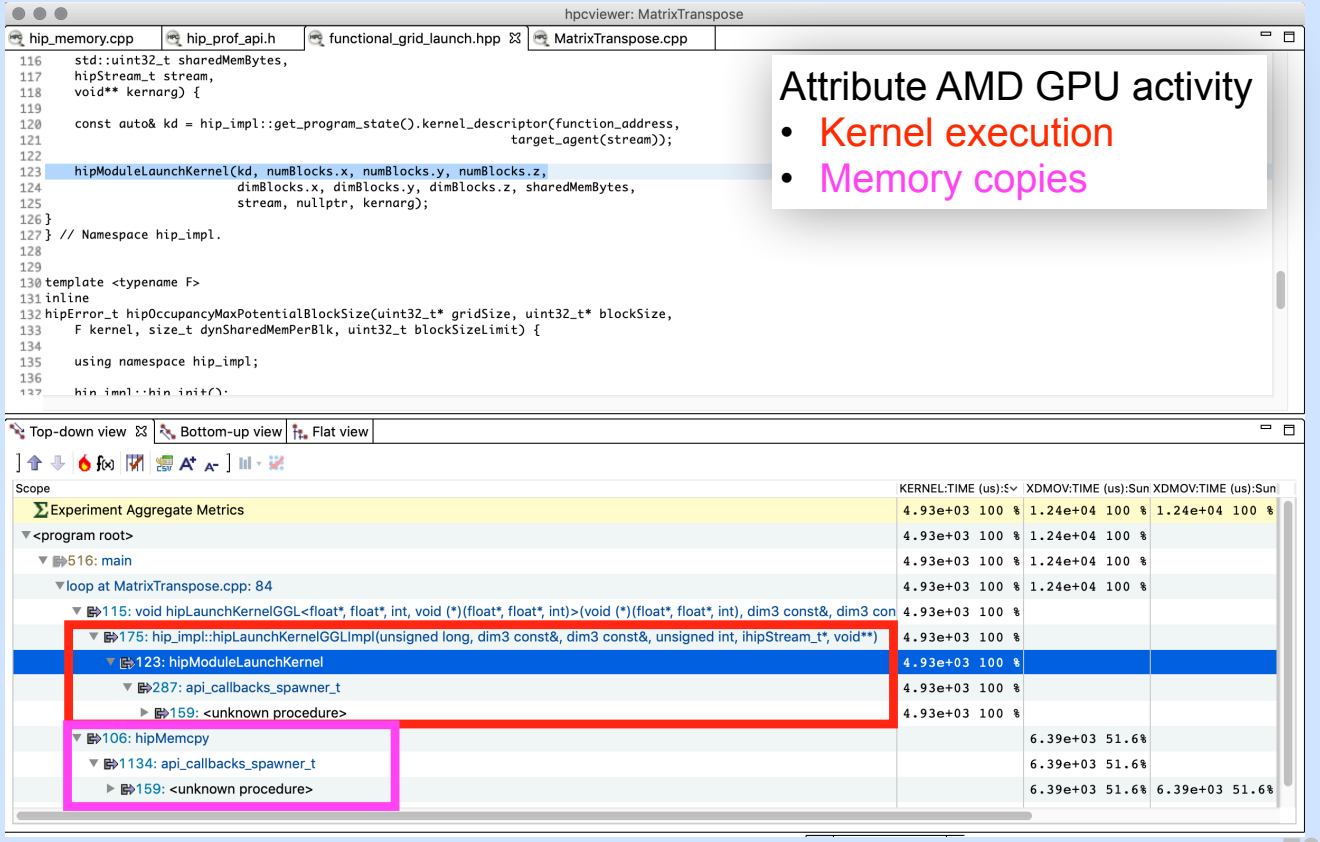
Reconstruct full calling contexts that include

- Inlined Kokkos templates
- Offloaded Kokkos CUDA computation

Prototype Integration with AMD's Roctracer GPU Monitoring Framework

- Use AMD Roctracer activity API to trace GPU activity
  - kernel launches
  - explicit memory copies
- Current prototype supports AMD's HIP programming model

AMD MatrixTranspose Testcase for Roctracer (AMD Ryzen + AMD 580 GPU)



HPCToolkit Challenges and Limitations

- **Fine-grain measurement and attribution of GPU performance**
  - PC sampling overhead on NIVIDIA GPUs is currently very high: a function of NVIDIA's CUPTI implementation
  - No available hardware support for fine-grain measurement on Intel and AMD GPUs
- **GPU tracing in HPCToolkit**
  - Creates one tool thread per GPU stream when tracing
  - OK for a small number of streams but many streams can be problematic
- **Cost of call path sampling**
  - Call path unwinding of GPU kernel invocations is costly (~2x execution dilation for Laghos)
  - Best solution is to avoid some of it, e.g. sample GPU kernel invocations
- **Currently, hpcprof and hpcprof-mpi compute dense vectors of metrics**
  - Designed for few CPU metrics, not O(100) GPU metrics: space and time problem for analysis

Analysis and Optimization Case Studies

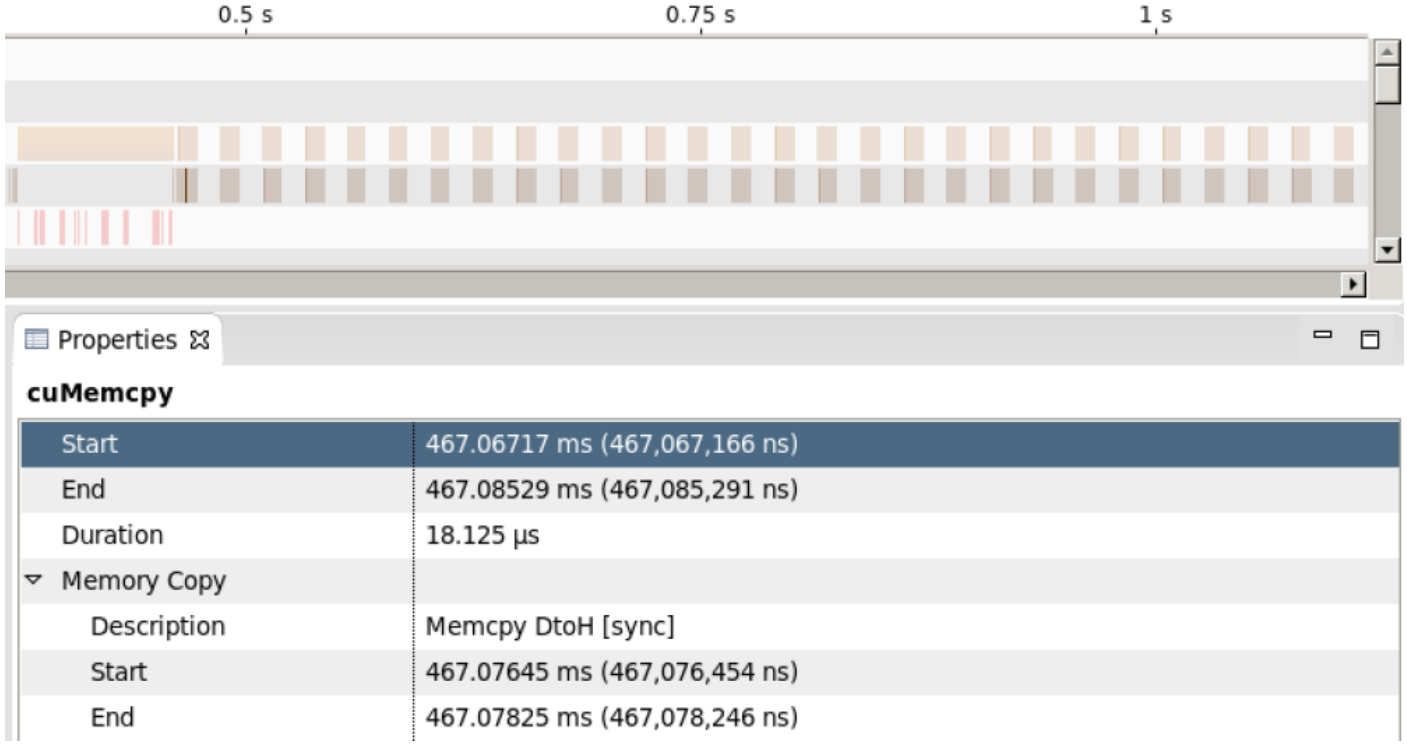
- **Environments**
  - Summit
    - cuda/10.1.168
    - gcc/6.4.0
  - Local
    - cuda/10.1.168
    - gcc/7.3.0

Case 1: Locating expensive GPU APIs with profile view

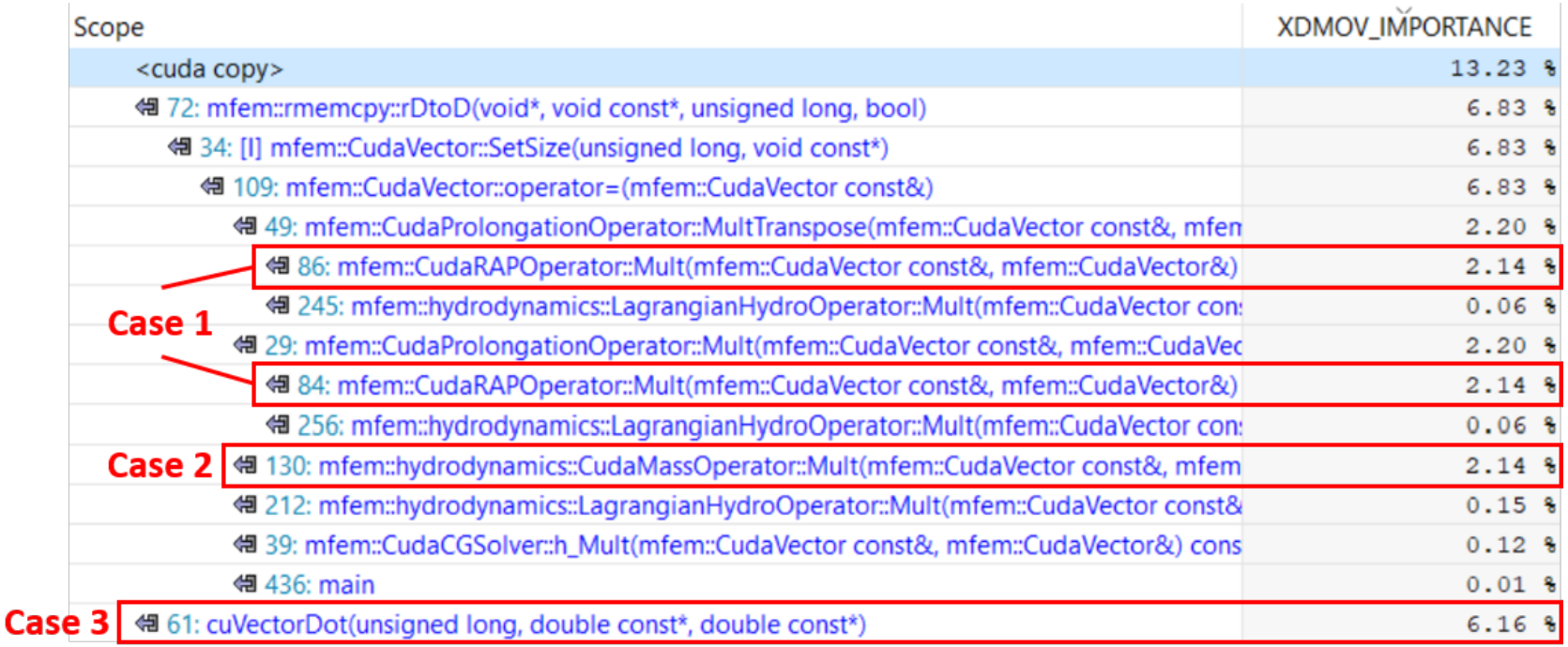
- **Laghos**
  - 1 MPI process
  - 1 GPU stream per process

nvprof: missing CPU calling context

- **Goal: Associate every GPU API with its CPU calling context**



Context-aware optimizations



Performance insight: Pin host memory page

- **A small amount of memory is transferred from device to host each time, repeated 197000 times**

Scope	GXCOPY (s):Sum (I)	GXCOPY:COUNT:Sum (I)	GXCOPY:D2H (B):Sum (I)
61: cuVectorDot(unsigned long, double const*, double const*)	3.67e-01 46.3%	1.97e+05 37.9%	7.81e+06 20.4%

- **Avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory**
  - Use pinned memory when data movement frequency is high but size is small

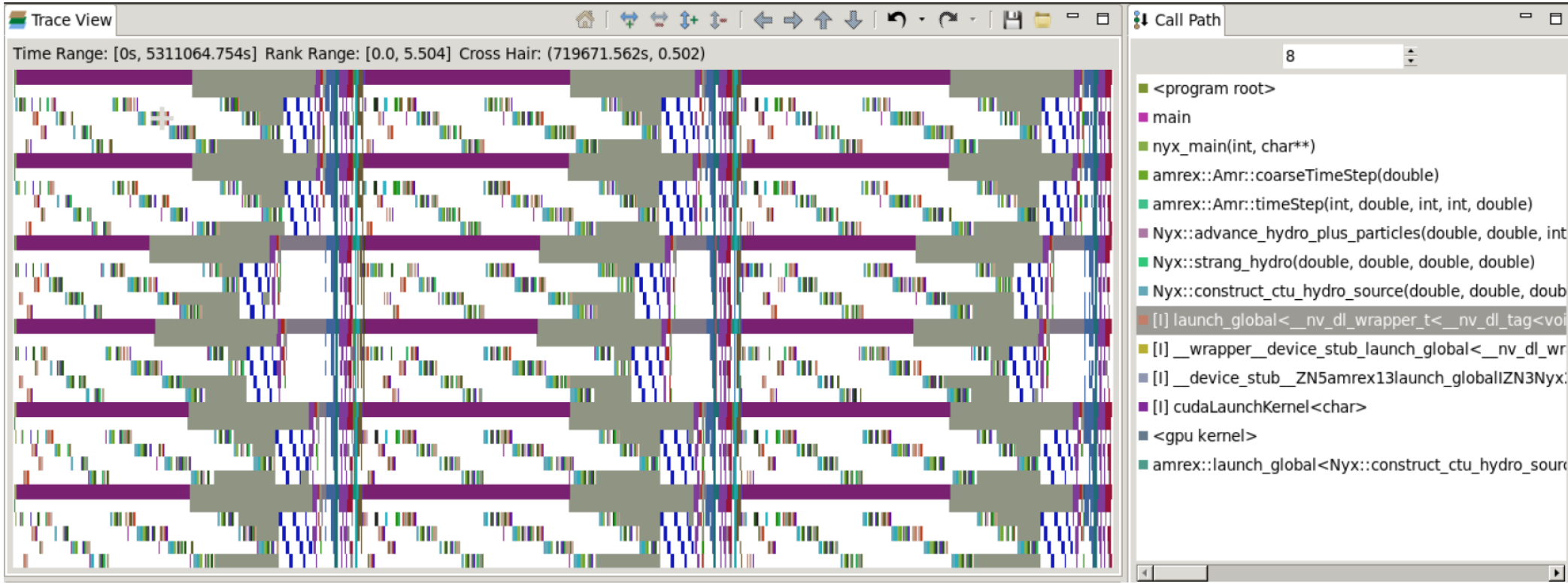
Case 2: Trace Applications at Large-scale

- **Nyx**
  - 6 MPI processes
  - 16 GPU stream per process
- **DCA++**
  - 60 MPI processes
  - 128 GPU stream per process

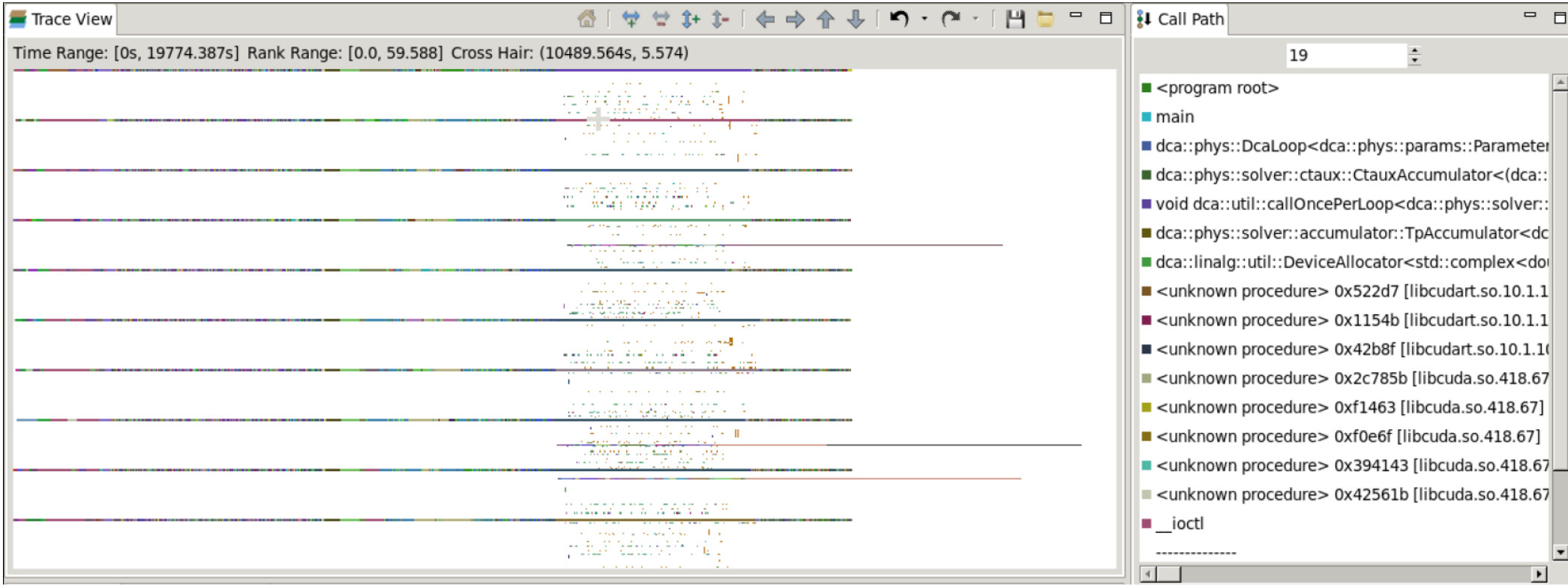
nvprof: Non-scalable Tracing of DCA++

- **nvprof**
  - With CPU profiling enabled, hangs on Summit
  - Without CPU profiling
    - Collects 1.1 GB data
- **Hpctoolkit**
  - CPU+GPU hybrid profiling with full calling context
    - Collects 0.13 GB data
    - Data can be further reduced by sampling GPU events

Nyx trace view

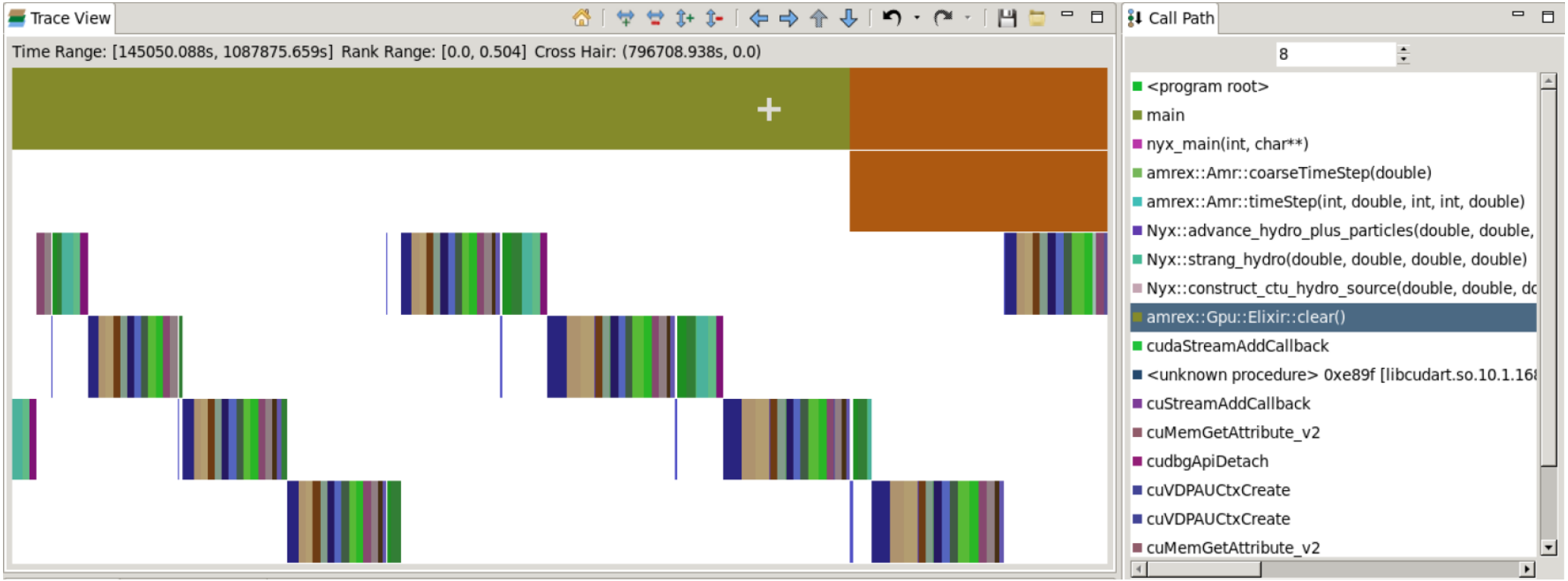


DCA++ trace view



Nyx insufficient GPU stream parallelism

- **On GPU, streams are not working concurrently**



Nyx cudaCallBack issue

- **On CPU, amrex::Gpu::Exlixir::clear() invokes stream callbacks**

```
33 void
34 Elixir::clear () noexcept
35 {
36 #ifdef AMREX_USE_GPU
37     if (Gpu::inLaunchRegion())
38     {
39         if (m_p != nullptr) {
40             void** p = static_cast<void**>(std::malloc(2*sizeof(void*)));
41             p[0] = m_p;
42             p[1] = (void*)m_arena;
43             AMREX_HIP_OR_CUDA(
44                 AMREX_HIP_SAFE_CALL ( hipStreamAddCallback(Gpu::gpuStream(),
45                                                            amrex_elixir_delete, p, 0));,
46                 AMREX_CUDA_SAFE_CALL(cudaStreamAddCallback(Gpu::gpuStream(),
47                                                            amrex_elixir_delete, p, 0)););
48             Gpu::callbackAdded();
49         }
50     }
51     else
52 #endif
```

Nyx performance insight

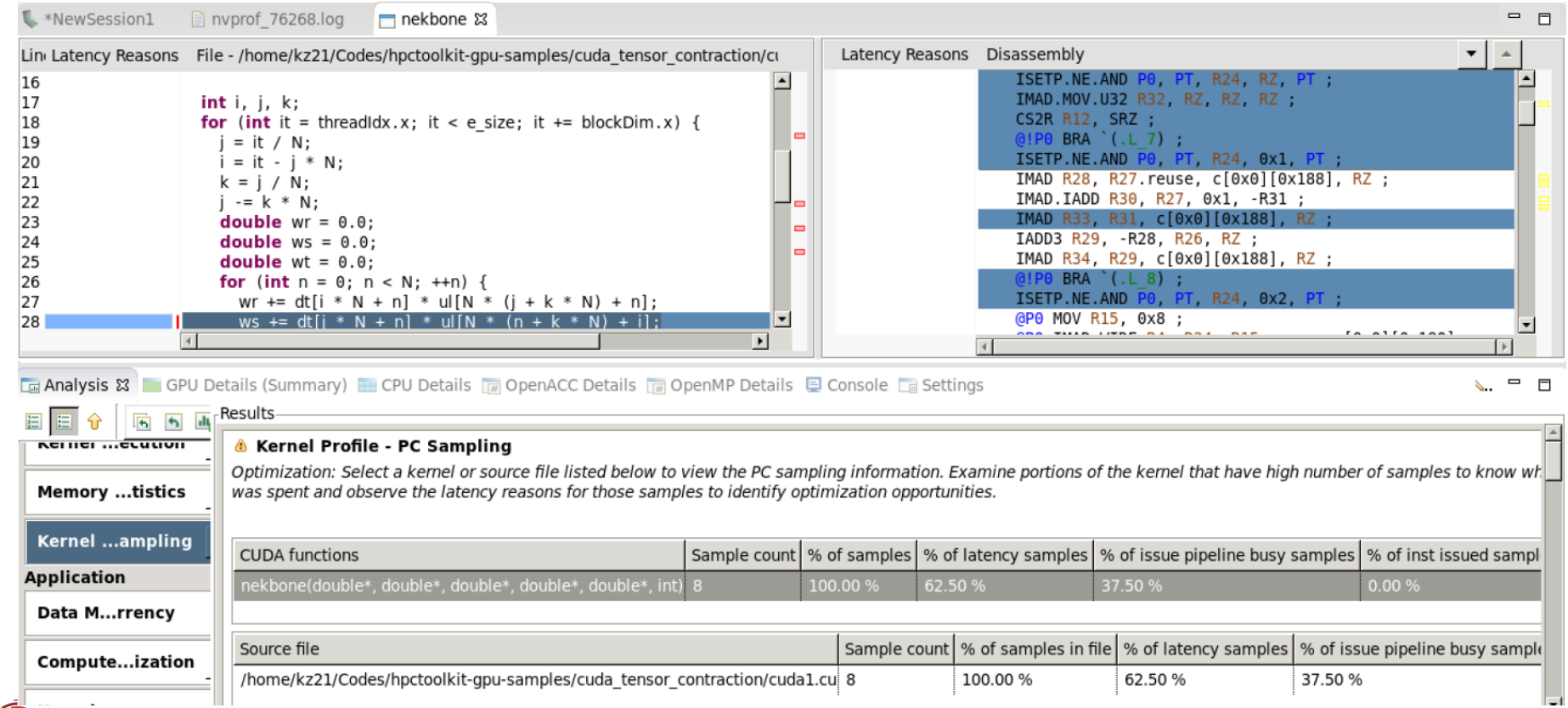
- **A bug present in the current version of CUDA (10.1). If a callBack is called in a place where multiple streams are used, the device kernels artificially synchronize and have no overlap.**
- **Fixed in CUDA-10.2?**
- **Workaround**
  - The Elixir object holds a copy of the data pointer to prevent it from being destroyed before the related device kernels are completed
  - Allocate new objects outside the compute loop and delete them after the completion of the work

Case 3: Fine-grained GPU Kernel Tuning

- **Nekbone: A lightweight subset of Nek5000 that mimics the essential computational complexity of Nek5000**

nvprof: Limited source level performance metrics

- **No loop structure, No GPU calling context, No instruction mix**



Nekbone Profile View

cuda1.cu | gpu-op-placeholders.c

```
16
17  int i, j, k;
18  for (int it = threadIdx.x; it < e_size; it += blockDim.x) {
19    j = it / N;
20    i = it - j * N;
21    k = j / N;
22    j -= k * N;
23    double wr = 0.0;
```

Top-down view | Bottom-up view | Flat view

Scope	GINS:Sum (I)	GINS:Sum (E)
516: main	6.59e+08 100 %	
150: [I] nekbone	6.59e+08 100 %	
2: __device_stub__Z7nekbonePdS_S_S_S_i(double*, double*, double*, double*, double*, int)	6.59e+08 100 %	
13: [I] cudaLaunchKernel<char>	6.59e+08 100 %	
209: <gpu kernel>	6.59e+08 100 %	
174: nekbone(double*, double*, double*, double*, double*, int)	6.59e+08 100 %	6.59e+08 100 %
loop at cuda1.cu: 18	3.17e+08 48.1%	3.17e+08 48.1%
loop at cuda1.cu: 39	2.21e+08 33.6%	2.21e+08 33.6%
loop at cuda1.cu: 11	6.47e+07 9.8%	6.47e+07 9.8%
cuda1.cu: 39	3.30e+07 5.0%	3.30e+07 5.0%
cuda1.cu: 11	1.31e+07 2.0%	1.31e+07 2.0%
cuda1.cu: 15	2.76e+06 0.4%	2.76e+06 0.4%

Performance insight 1: Execution dependency

- **The hotspot statement is waiting for *j* and *k***

cuda1.cu

```
14
15  __syncthreads();
16
17  int i, j, k;
18  for (int it = threadIdx.x; it < e_size; it += blockDim.x) {
19    j = it / N;
20    i = it - j * N;
21    k = j / N;
22    j -= k * N;
23    double wr = 0.0;
24    double ws = 0.0;
25    double wt = 0.0;
26    for (int n = 0; n < N; ++n) {
27      wr += dt[i * N + n] * ul[N * (j + k * N) + n];
28      ws += dt[j * N + n] * ul[N * (n + k * N) + i];
29      wt += dt[k * N + n] * ul[N * (j + n * N) + i];
30    }
```

Top-down view | Bottom-up view | Flat view

Scope	GINS:Sum (I)	GINS:Sum (E)	GINS:STL_ANY:Sl	GINS:STL_ANY:Sl	GI
209: <gpu kernel>	6.59e+08 100 %		3.70e+08 100 %		3.0
174: nekbone(double*, double*, double*, double*, double*, int)	6.59e+08 100 %	6.59e+08 100 %	3.70e+08 100 %	3.70e+08 100 %	3.0
loop at cuda1.cu: 18	3.17e+08 48.1%	3.17e+08 48.1%	1.79e+08 48.3%	1.79e+08 48.3%	6.1
cuda1.cu: 27	8.80e+07 13.4%	8.80e+07 13.4%	4.92e+07 13.3%	4.92e+07 13.3%	1.1
cuda1.cu: 32	7.72e+07 11.7%	7.72e+07 11.7%	5.36e+07 14.5%	5.36e+07 14.5%	
cuda1.cu: 28	5.95e+07 9.0%	5.95e+07 9.0%	3.25e+07 8.8%	3.25e+07 8.8%	
cuda1.cu: 29	5.19e+07 7.9%	5.19e+07 7.9%	2.95e+07 8.0%	2.95e+07 8.0%	

Strength reduction

- **MISC.CONVERT: I2F, F2I, MUFU instructions**
  - NVIDIA GPUs convert integer to float for division
  - High latency and low throughput instruction
- **Replace *j = it / N* by *j = it x (1/N)* and precompute *1/N***

```
18  for (int it = threadIdx.x; it < e_size; it += blockDim.x) {
19    j = it / N;
20    i = it - j * N;
21    k = j / N;
22    j -= k * N;
23    double wr = 0.0;
24    double ws = 0.0;
25    double wt = 0.0;
26    for (int n = 0; n < N; ++n) {
27      wr += dt[i * N + n] * ul[N * (j + k * N) + n];
```

Top-down view | Bottom-up view | Flat view

Scope	MISC.CONVERT:Sum (I)	MISC.CONVERT:Sum (E)
174: nekbone(double*, double*, d	2.01e+05 100 %	2.01e+05 100 %
loop at cuda1.cu: 18	1.02e+05 51.0%	1.02e+05 51.0%
cuda1.cu: 27		
cuda1.cu: 32		
cuda1.cu: 28		
cuda1.cu: 29		
cuda1.cu: 19	1.02e+05 51.0%	1.02e+05 51.0%

Coming Attraction: Instruction-level Analysis

Separate GPU instructions into classes

- **Memory operations**
  - instruction (load, store)
  - size
  - memory kind (global memory, texture memory, constant memory)
- **Floating point**
  - instruction (add, mul, mad)
  - size
  - compute unit (tensor unit, floating point unit)
- **Integer operations**
- **Control operations**
  - branches, calls

Performance insight 2: Instruction Throughput

- **Estimate instruction throughput based on pc samples**

- $THROUGHPUT = \frac{INS}{TIME}$

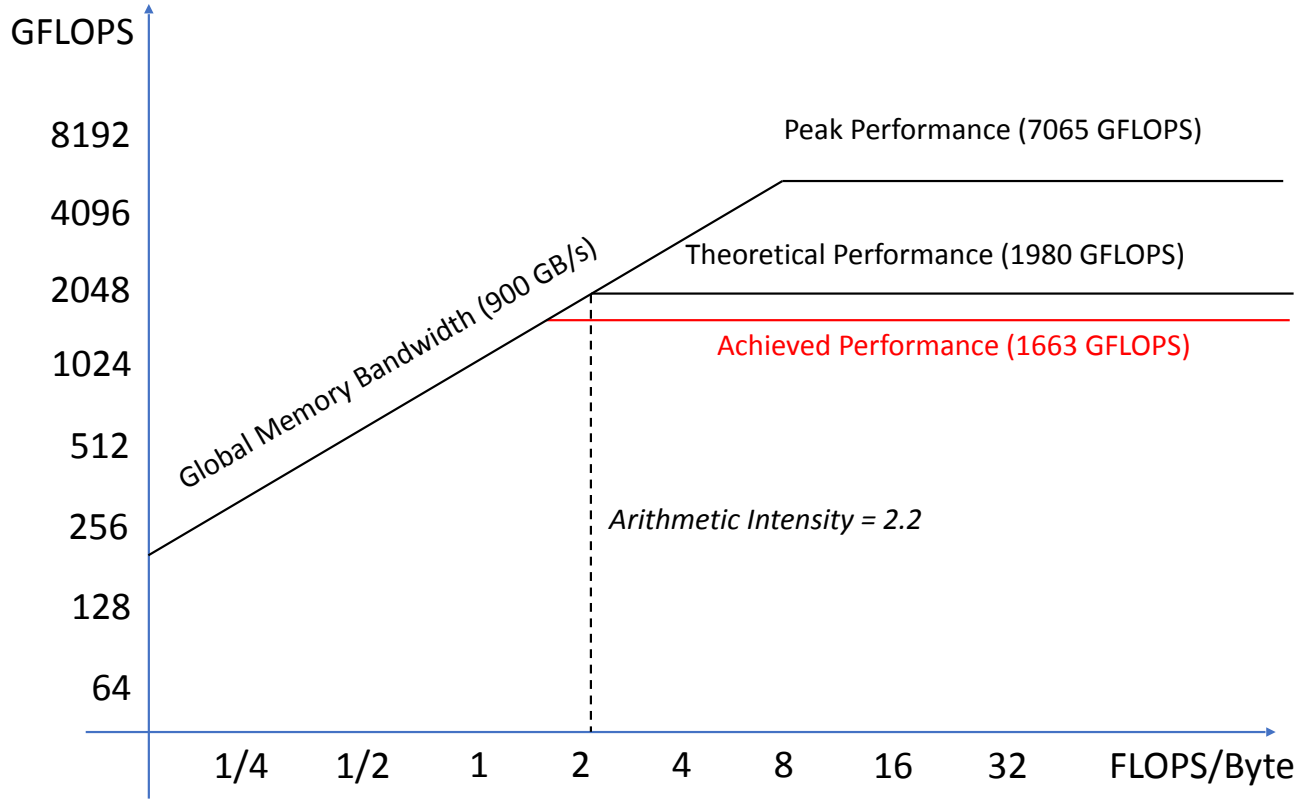
- $GFLOPS = THROUGHPUT_{DP}$

- $Arithmetic\ Intensity = \frac{THROUGHPUT_{GMEM}}{THROUGHPUT_{DP}}$

Scope	MEMORY.LOAD.GLOBAL.64	MEMORY.STORE.GLOBAL.64	FLOAT.MAD.64:Sum	FLOAT.MUL.64:Sum	FLOAT.ADD.64:Sum
<program root>	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
516: main	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
[I] inlined from cuda4.cu: 2	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
2: __device_stub__Z7nekboneP	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
[I] inlined from cuda_runtime.h	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
209: <gpu kernel>	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
174: nekbone(double*,	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %

Roofline analysis

- **83.9% of peak performance**



Performance insight 3: unfused DMUL and DADD

- **DMUL:** 6.51×10^5
- **DADD:** 4.55×10^5
- **If all paired DMUL and DADD instructions are fused to MAD instructions**
  - $$\frac{\left(4.55 \times 10^5 + 3.08 \times 10^6\right)}{3.08 \times 10^6} = 14.7\%$$
  - 1663 GFLOPS × 114.7% = 1908 GFLOPS (99% of peak)

Scope	MEMORY.LOAD.GLOBAL.64	MEMORY.STORE.GLOBAL.64	FLOAT.MAD.64:Sum	FLOAT.MUL.64:Sum	FLOAT.ADD.64:Sum
<program root>	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
516: main	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
[I] inlined from cuda4.cu: 2	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
2: __device_stub__Z7nekboneP	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
[I] inlined from cuda_runtime.h	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
209: <gpu kernel>	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %
174: nekbone(double*,	3.36e+05 100 %	5.32e+04 100 %	3.08e+06 100 %	6.51e+05 100 %	4.55e+05 100 %

Case Study Acknowledgements

- **ORNL**
  - Ronnie Chatterjee
- **IBM**
  - Eric Liu
- **NERSC**
  - Christopher Daley
  - Jean Sexton
  - Kevin Gott

Installing HPCToolkit for Analysis of GPU-accelerated Codes

- **Full instructions:** http://hpctoolkit.org/software-instructions.html
- **The short form**
  - Clone spack
    - command: **`git clone https://github.com/spack/spack`**
  - Configure a packages.yaml file
    - specify your platform's installation of CUDA or ROCM
    - specify your platform's installation of MPI
    - use an appropriate GCC compiler
      - ensure that a GCC version >= 5 is on your path. typically, we use GCC 7.3
      - **`spack compiler find`**
  - Install software for your platform using spack
    - NVIDIA GPUs: **`spack install hpctoolkit@master +cuda +mpi`**
    - AMD GPUs: **`spack install hpctoolkit@master +rocm +mpi`**